NO.36

编程狂人

Programming Madman

推酷

关于推酷

推酷是专注于IT圈的个性化阅读社区。我们利用智能算法, 从海量文章资讯中挖掘出高质量的内容,并通过分析用户 的阅读偏好,准实时推荐给你最感兴趣的内容。我们推荐 的内容包含科技、创业、设计、技术、营销等内容,满足 你日常的专业阅读需要。我们针对IT人还做了个活动频道, 它聚合了IT圈最新最全的线上线下活动,使IT人能更方便地 找到感兴趣的活动信息。

关于周刊

《编程狂人》是献给广大程序员们的技术周刊。我们利用技术挖掘出那些高质量的文章,并通过人工加以筛选出来。每期的周刊一般会在周二的某个时间点发布,敬请关注阅读。

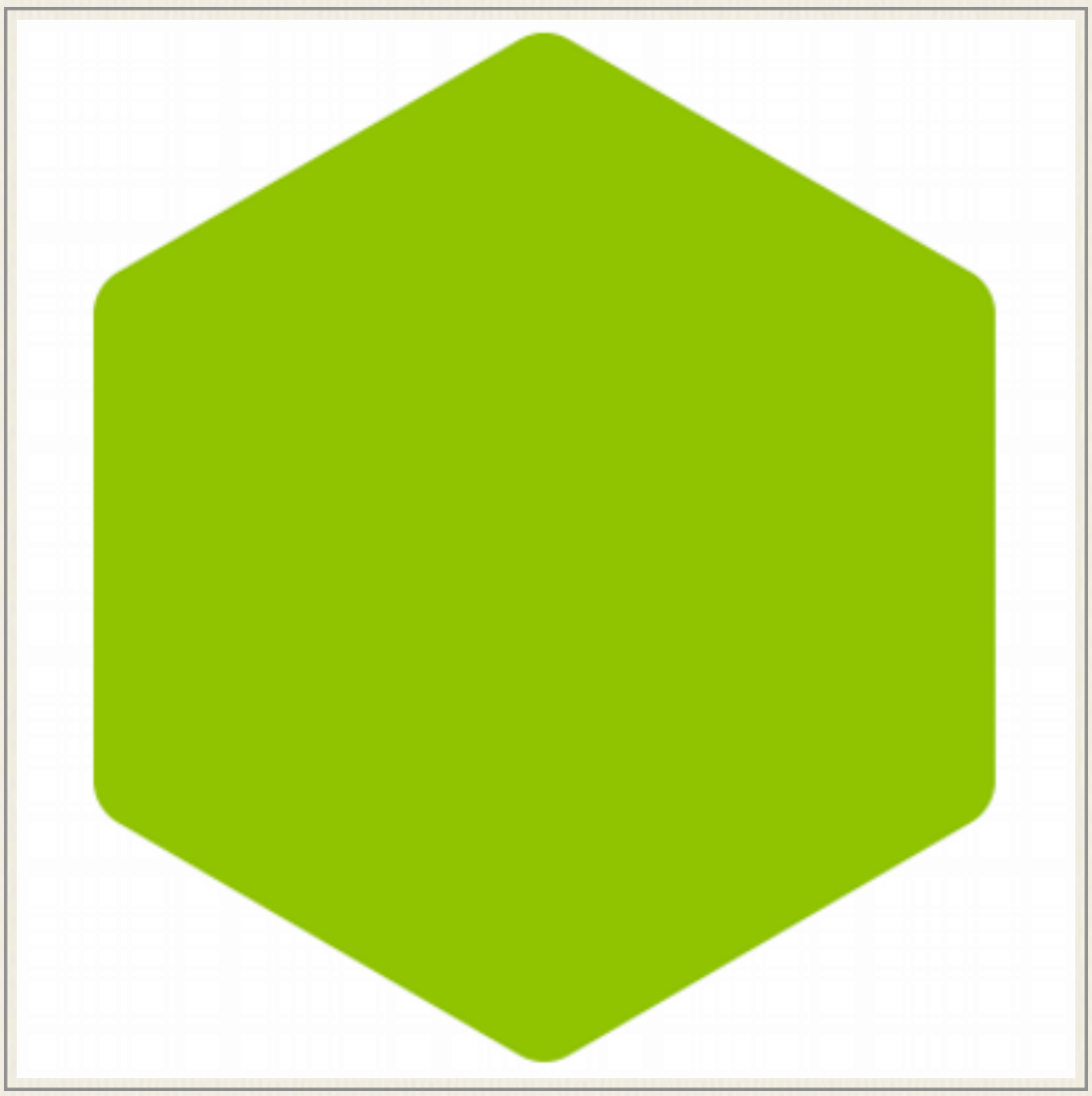
本期为精简版 周刊完整版链接:http://www.tuicool.com/mags/53df98b5d91b146d9103074f

欢迎下载推酷客户端体验更多阅读乐趣

目录

Node Die, Node Rebirth

作者：红豆志



最近一段时间 Node 社区发生了太多太多的事情. 先是 Ben Noordhuis 因为人称代词风波离开, 后是 Isaac Schlueter 因为忙于 npm,Inc. 发展将大旗交给了 TJ Fontaine. 再是社区大神, Express 作者 TJ Holowaychuk 因为一些原因离开 Node 社区, 转而去写 Go. 再加上 0.12 迟迟没有发布, 某些项目放弃 Node 转而使用其他技术, 让许多开发者不禁纳闷 Node 到底是怎么了, 难道开始陨落了么?

Node Die

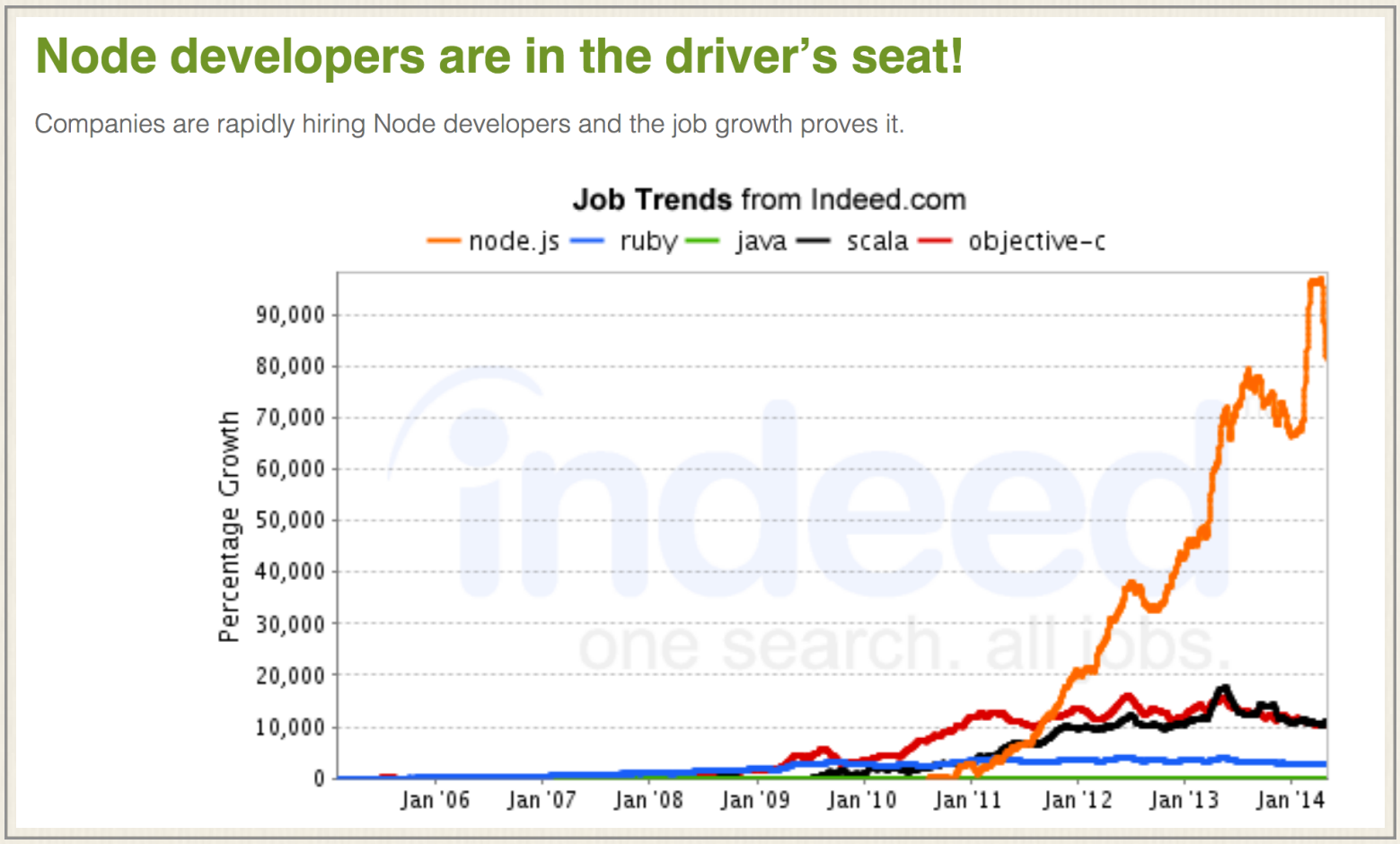
Node Die 不是因为 TJ 大神离开, Node Die 不是说 Node 生态出问题 (Node 生态非常健康), 而是作为开源项目的 Node 已经死去, 现在的 Node 被牢牢的握在 Joyent 手里(更新后的官网Node.js标识已打上了®标识), 而且已经开始尝试通过 Node 盈利; Node 生态的重要组成部分 NPM 也被 Isaac 拿来做盈利探索; 使用最多, 最有名的框架 Express 被移交给了 StrongLoop (最早的以Node作为创业的公司).

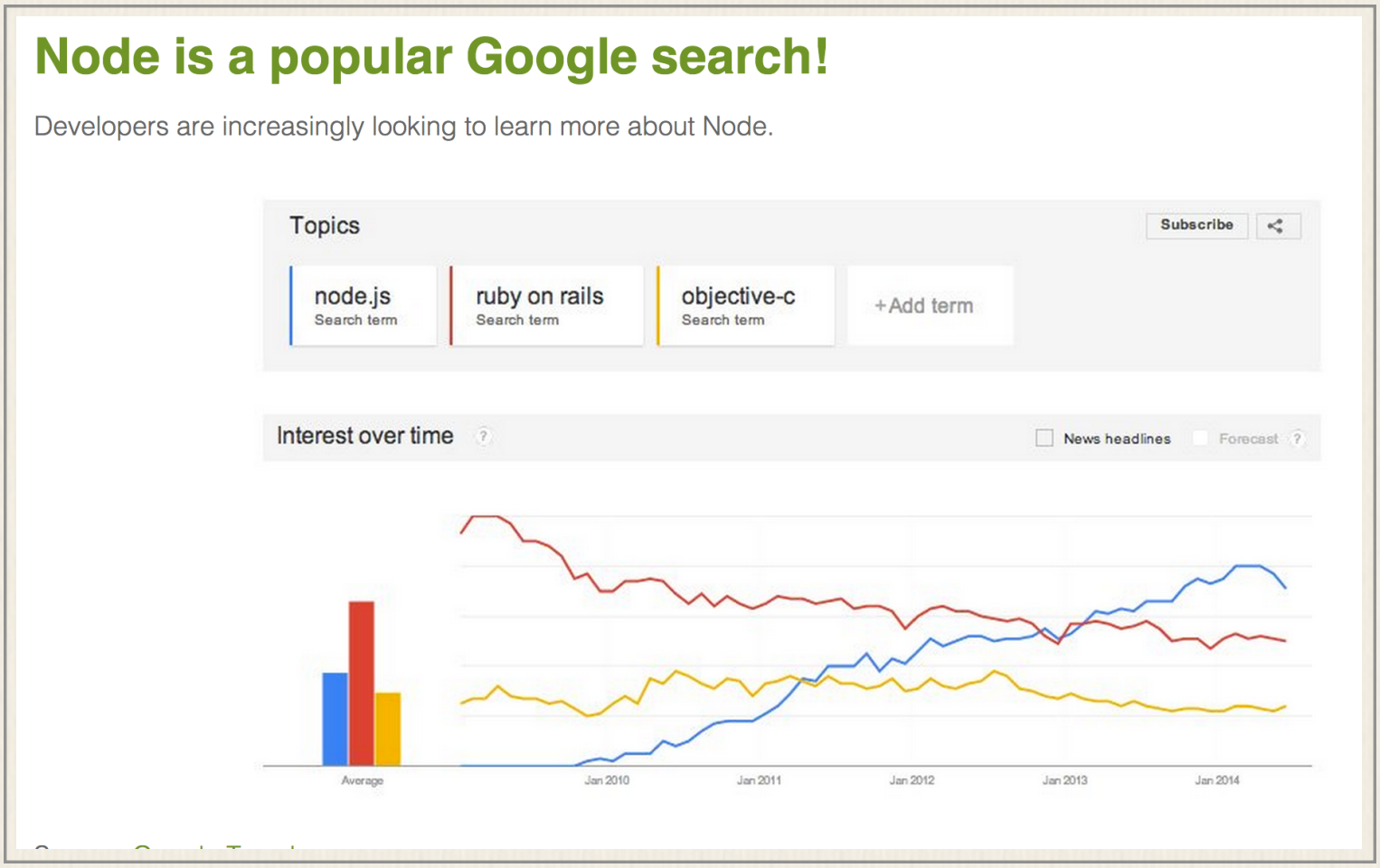
TJ 大神的离开的确是社区的一大损失, 同时也提醒我们 Node 不是圣杯, 并不适合所有场合, 同时 JS 语言本身存在巨大问题

Node Rebirth

我们使用 Node 是因为有它非常适合的场合: Web 开发, API 开发, Internet of Thing 等.

Node 社区发展非常健康, 各个国家大会不是举办, NPM 成为最大的包管理平台(数量), 许多大公司开始使用 Node, Node 开发者招聘飞快增长, Node 关注度不断提高. Node 不仅催生了 bower, grunt 等一匹工具, 还催生了 StrongLoop, NodeSource, Ghost, NodeBB 等一匹创业公司





商业化的 Node 并不是一无是处, 商业化一样可以保持它的活力, 商业化可以促进 Node 和 NPM 持续发展, 提供更快,更稳定的 NPM 服务.

New Team

作为 Node 新任掌门人 TJ Fontaine 最大的贡献是 Walmart Node 内存泄露问题诊断和解除. 之后他便开始了漫长的 Node.js on the road 活动, 用于收集开发者对 Node 的使用经验和问题反馈. 也让我们经历了漫长的 0.12 等待. 活动归来之后他做出了一个非常明智的决定: 修改 Node 的授权协议, 让更多人能容易参与贡献. 并重新改版了官方网站, 将文档和社区参与度最为重要优化部分. 使得 Node 官网更加规范, 摆脱以前玩具的印象.

从 TJ Fontaine 接管 Node 后的一系列行动, 可以看出它的确喜欢 Node, 真心想推动 Node 不断向前发展.

让我们耐下心来等待 0.12, 等待更美好的 Node 到来.

原文链接：http://blog.rednode.cn/thoughts-on-node-js/

Angularjs学习笔记

作者：邹业盛

1. 关于AngularJS

AngularJS 是 Google 开源出来的一套 js 工具。下面简称其为 ng 。这里只说它是“工具”，没说它是完整的“框架”，是因为它并不是定位于去完成一套框架要做的事。更重要的，是它给我们揭示了一种新的应用组织与开发方式。

ng 最让我称奇的，是它的数据双向绑定。其实想想，我们一直在提数据与表现的分离，但是这里的“双向绑定”从某方面来说，是把数据与表现完全绑定在一起——数 据变化，表现也变化。反之，表现变化了，内在的数据也变化。有过开发经验的人能体会到这种机制对于前端应用来说，是很有必要的，能带来维护上的巨大优势。 当然，这里的绑定与提倡的分离并不是矛盾的。

ng 可以和 jQuery 集成工作，事实上，如果没有 jQuery ， ng 自己也做了一个轻量级的 jQuery ，主要实现了元素操作部分的 API 。

关于 ng 的几点：

- 对 IE 方面，它兼容 IE8 及以上的版本。
- 与 jQuery 集成工作，它的一些对象与 jQuery 相关对象表现是一致的。
- 使用 ng 时不要冒然去改变相关 DOM 的结构。

2. 关于本文档

这份文档如其名，是我自己学习 ng 的过程记录。只是过程记录，没有刻意像教程那样去做。所以呢，从前至后，中间不免有一些概念不清不明的地

方。因为事实上，在某个阶段对于一些概念本来就不 可能明白。所以，整个过程只求在形式上的能用即可——直到最后的“自定义”那几章，特别是“自定义指令”，那几章过完，你才能看清 ng 本来的面貌。前面就不要太纠结概念，本质，知道怎么用就好。

3. 开始的例子

我们从一个完整的例子开始认识 ng：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />

<title>试验</title>

<script type="text/javascript" src="jquery-1.8.3.js"></script>
<script type="text/javascript" src="angular.js"></script>

</head>
<body>
  <div ng-controller="BoxCtrl">
    <div style="width: 100px; height: 100px; background-color: red;"
         ng-click="click()"></div>
    <p>{{ w }} x {{ h }}</p>
    <p>W: <input type="text" ng-model="w" /></p>
    <p>H: <input type="text" ng-model="h" /></p>
  </div>


<script type="text/javascript" charset="utf-8">


var BoxCtrl = function($scope, $element){

  //$element 就是一个 jQuery 对象
  var e = $element.children().eq(0);
  $scope.w = e.width();
  $scope.h = e.height();

  $scope.click = function(){
    $scope.w = parseInt($scope.w) + 10;
    $scope.h = parseInt($scope.h) + 10;
  }
```

```
36
37      $scope.$watch('w',
38        function(to, from){
39          e.width(to);
40        }
41      );
42
43      $scope.$watch('h',
44        function(to, from){
45          e.height(to);
46        }
47      );
48    }
49
50    angular.bootstrap(document.documentElement);
51    </script>
52    </body>
53    </html>
```

从上面的代码中，我们看到在通常的 HTML 代码当中，引入了一些标记，这些就是 ng 的模板机制，它不光完成数据渲染的工作，还实现了数据绑定的功能。

同时，在 HTML 中的本身的 DOM 层级结构，被 ng 利用起来，直接作为它的内部机制中，上下文结构的判断依据。比如例子中 p 是 div 的子节点，那么 p 中的那些模板标记就是在 div 的 Ctrl 的作用范围之内。

其它的，也同样写一些 js 代码，里面重要的是作一些数据的操作，事件的绑定定义等。这样，数据的变化就会和页面中的 DOM 表现联系起来。一旦这种联系建立起来，也即完成了我们所说的“双向绑定”。然后，这里说的“事件”，除了那些“点击”等通常的 DOM 事件之外，我们还更关注“数据变化”这个事件。

最后，可以使用：

angular.bootstrap(document.documentElement);

来把整个页面驱动起来了。（你可以看到一个可被控制大小的红色方块）

更完整的方法是定义一个 APP：

```
<!DOCTYPE html>
<html ng-app="MyApp">
<head>
<meta charset="utf-8" />

<title>数据正向绑定</title>

<script type="text/javascript" src="jquery-1.8.3.js"></script>
<script type="text/javascript" src="angular.js"></script>

</head>
<body>

<div ng-controller="TestCtrl">
  <input type="text" value="" id="a" />
</div>


<script type="text/javascript">
var TestCtrl = function(){
  console.log('ok');
}

//angular.bootstrap(document.documentElement);
angular.module('MyApp', [], function(){console.log('here')});
</script>

</body>
</html>
```

这里说的一个 App 就是 ng 概念中的一个 Module 。对于 Controller 来说， 如果不想使用全局函数，也可以在 app 中定义：

```
var app = angular.module('MyApp', [], function(){console.log('here')});
app.controller('TestCtrl',
  function($scope){
    console.log('ok');
  }
);
```

上面我们使用 ng-app 来指明要使用的 App ， 这样的话可以把显式的初始化工作省了。一般完整的过程是：

```
var app = angular.module('Demo', [], angular.noop);
angular.bootstrap(document, ['Demo']);
```

使用 angular.bootstrap 来显示地做初始化工具，参数指明了根节点，装载的模块（可以是多个模块）。

4. 依赖注入

injector ， 我从 ng 的文档中得知这个概念，之后去翻看源码时了解了一下这个机制的工作原理。感觉就是虽然与自己的所想仅差那么一点点，但就是这么一点点，让我感慨想象力之神奇。

先看我们之前代码中的一处函数定义：

var BoxCtrl = function($scope, $element){}

在这个函数定义中，注意那两个参数： $scope ， $element ，这是两个很有意思的东西。总的来说，它们是参数，这没什么可说的。但又不仅仅是参数——你换个名字代码就不能正常运行了。

事实上，这两个参数，除了完成“参数”的本身任务之外，还作为一种语法糖完成了“依赖声明”的任务。本来这个函数定义，完整的写法应该像 AMD 声明一样，写成：

var BoxCtrl = ['$scope', '$element', function(s, e){}];

这样就很明显，表示有一个函数，它依赖于两个东西，然后这两个东西会依次作为参数传入。

简单起见，就写成了一个函数定义原本的样子，然后在定义参数的名字上作文章，来起到依赖声明的作用。

在处理时，通过函数对象的 toString() 方法可以知道这个函数定义代码的字符串表现形式，然后就知道它的参数是 $scope 和 $element 。通过名字判断出这是两个外部依赖，然后就去获取资源，最后把资源作为参数，调用定义的函数。

所以，参数的名字是不能随便写的，这里也充分利用了 js 的特点来尽量做到“反省”了。

在 Python 中受限于函数名的命名规则，写出来不太好看。不过也得利于反省机制，做到这点也很容易：

```
# -*- coding: utf-8 -*-

def f(Ia, Ib):
    print Ia, Ib

args = f.func_code.co_varnames
SRV_MAP = {
    'Ia': '123',
    'Ib': '456',
}

srv = {}
for a in args:
    if a in SRV_MAP:
        srv[a] = SRV_MAP[a]
f(**srv)
```

5. 作用域

这里提到的“作用域”的概念，是一个在范围上与 DOM 结构一致，数据上相对于某个 $scope 对象的属性的概念。我们还是从 HTML 代码上来入手：

```
<div ng-controller="BoxCtrl">
  <div style="width: 100px; height: 100px; background-color: red;"
       ng-click="click()">
  </div>
  <p>{{ w }} x {{ h }}</p>
  <p>W: <input type="text" ng-model="w" /></p>
  <p>H: <input type="text" ng-model="h" /></p>
</div>
```

上面的代码中，我们给一个 div 元素指定了一个 BoxCtrl ， 那么， div 元素之内，就是 BoxCtrl 这个函数运行时， $scope 这个注入资源的控制范围。在代码中我们看到的 click() ， w ， h 这些东西，它们本来的位置对应于 $scope.click ， $scope.w ， $scope.h 。

我们在后面的 js 代码中，也可以看到我们就是在操作这些变量。依赖于 ng 的数据绑定机制，操作变量的结果直接在页面上表现出来了。

6. 数据绑定与模板

我纠结了半天，“数据绑定”与“模板”这两个东西还真没办法分开来说。因为数据绑定需要以模板为载体，离开了模板，数据还绑个毛啊。

ng 的一大特点，就是数据双向绑定。双向绑定是一体，为了描述方便，下面分别介绍。

6.1. 数据->模板

数据到表现的绑定，主要是使用模板标记直接完成的：

```
<p>{{ w }} x {{ h }}</p>
```

使用 {{ }} 这个标记，就可以直接引用，并绑定一个作用域内的变量。在实现上， ng 自动创建了一个 watcher 。效果就是，不管因为什么，如果作用域的变量发生了改变，我们随时可以让相应的页面表现也随之改变。我们可以看一个更纯粹的例子：

```
<p id="test" ng-controller="TestCtrl">{{ a }}</p>

<script type="text/javascript">
var TestCtrl = function($scope){
  $scope.a = '123';
}
angular.bootstrap(document.documentElement);
```

上面的例子在页面载入之后，我们可以在页面上看到 123 。这时，我们可以打开一个终端控制器，输入：

```
$('#test').scope().a = '12345';

$('#test').scope().$digest();
```

上面的代码执行之后，就可以看到页面变化了。

对于使用 ng 进行的事件绑定，在处理函数中就不需要去关心 $digest() 的调用了。因为 ng 会自己处理。源码中，对于 ng 的事件绑定，真正的处理函数不是指定名字的函数，而是经过 $apply() 包装过的一个函数。这个 $apply() 做的一件事，就是调用根作用域 $rootScope 的 $digest() ，这样整个世界就清净了：

```
<p id="test" ng-controller="TestCtrl" ng-click="click()">{{ a }}</p>


<script type="text/javascript" charset="utf-8">
var TestCtrl = function($scope){
  $scope.a = '123';

  $scope.click = function(){
    $scope.a = '456';
  }
}
angular.bootstrap(document.documentElement);
```

那个 click 函数的定义，绑定时变成了类似于：

```
function(){
  $scope.$apply(
    function(){
```

```
        $scope.click();
      }
    )
  }
```

这里的 $scope.$apply() 中做的一件事：

```
  $rootScope.$digest();
```

6.2. 模板->数据

模板到数据的绑定，主要是通过 ng-model 来完成的：

```
  <input type="text" id="test" ng-controller="TestCtrl" ng-model="a" />

  <script type="text/javascript" charset="utf-8">
  var TestCtrl = function($scope){
    $scope.a = '123';
  }
```

这时修改 input 中的值，然后再在控制终端中使用：

```
  $('#test').scope().a
```

查看，发现变量 a 的值已经更改了。

实际上， ng-model 是把两个方向的绑定都做了。它不光显示出变量的值，也把显示上的数值变化反映给了变量。这个在实现上就简单多了，只是绑定 change 事件，然后做一些赋值操作即可。不过 ng 里，还要区分对待不同的控件。

6.3. 数据->模板->数据->模板

现在要考虑的是一种在现实中很普遍的一个需求。比如就是我们可以输入数值，来控制一个矩形的长度。在这里，数据与表现的关系是：

- 长度数值保存在变量中
- 变量显示于某个 input 中
- 变量的值即是矩形的长度
- input 中的值变化时，变量也要变化
- input 中的值变化时，矩形的长度也要变化

当然，要实现目的在这里可能就不止一种方案了。按照以前的做法，很自然地会想法，绑定 input 的 change 事件，然后去做一些事就好了。但是，我们前面提到过 ng-model 这个东西，利用它就可以在不手工处理 change 的条件下完成数据的展现需求，在此基础之上，我们还需要做的一点，就是把变化后的数据应用到矩形的长度之上。

最开始，我们面对的应该是这样一个东西：

```
<div ng-controller="TestCtrl">
  <div style="width: 100px; height: 10px; background-color: red"></
div>
  <input type="text" name="width" ng-model="width" />
</div>

<script type="text/javascript" charset="utf-8">
var TestCtrl = function($scope){
  $scope.width = 100;
}
angular.bootstrap(document.documentElement);
</script>
```

我们从响应数据变化，但又不使用 change 事件的角度来看，可以这样处理宽度变化：

```
var TestCtrl = function($scope, $element){
  $scope.width = 100;
  $scope.$watch('width',
    function(to, from){
      $element.children(':first').width(to);
    }
  );
}
```

使用 $watch() 来绑定数据变化。

当然，这种样式的问题，有更直接有效的手段， ng 的数据绑定总是让人惊异：

```
<div ng-controller="TestCtrl">
  <div style="width: 10px; height: 10px; background-color: red" ng-style="style">
  </div>
  <input type="text" name="width" ng-model="style.width" />
</div>


<script type="text/javascript" charset="utf-8">
var TestCtrl = function($scope){
  $scope.style = {width: 100 + 'px'};
}
```

angular.bootstrap(document.documentElement);

</script>

7. 模板

前面讲了数据绑定之后，现在可以单独讲讲模板了。

作为一套能称之谓“模板”的系统，除了能干一些模板的常规的事之外（好吧，即使是常规的逻辑判断现在它也做不了的），配合作用域 $scope 和 ng 的数据双向绑定机制， ng 的模板系统就变得比较神奇了。

7.1. 定义模板内容

定义模板的内容现在有三种方式：

1. 在需要的地方直接写字符串
2. 外部文件
3. 使用 script 标签定义的“内部文件”

第一种不需要多说。第二种和第三种都可以和 ng-include 一起工作，来引入一段模板。

直接引入同域的外部文件作为模板的一部分：

<div ng-include src="'tpl.html'">

* *

* *

* *

注意， src 中的字符串会作为表达式处理（可以是 $scope 中的变量），所以，直接写名字的话需要使用引号。

引入 script 定义的“内部文件”：

<script type="text/ng-template" id="tpl">

```
here, {{ 1 + 1 }}
</script>

<div ng-include src="'tpl'"></div>
```

配合变量使用:

```
<script type="text/ng-template" id="tpl">
here, {{ 1 + 1 }}
</script>

<a ng-click="v='tpl'">Load</a>
<div ng-include src="v"></div>
```

7.2. 内容渲染控制

7.2.1. 重复 ng-repeat

这算是唯一的一个控制标签么……，它的使用方法类型于:

```
<div ng-controller="TestCtrl">
  <ul ng-repeat="member in obj_list">
    <li>{{ member }}</li>
  </ul>
</div>


var TestCtrl = function($scope){
  $scope.obj_list = [1,2,3,4];
```

```
}
```

除此之外，它还提供了几个变量可供使用：

- $index 当前索引
- $first 是否为头元素
- $middle 是否为非头非尾元素
- $last 是否为尾元素

```
<div ng-controller="TestCtrl">
  <ul ng-repeat="member in obj_list">
    <li>{{ $index }}, {{ member.name }}</li>
  </ul>
</div>
```

```
var TestCtrl = function($scope){
  $scope.obj_list = [{name: 'A'}, {name: 'B'}, {name: 'C'}];
}
```

7.2.2. 赋值 ng-init

这个指令可以在模板中直接赋值，它作用于 angular.bootstrap 之前，并且，定义的变量与 $scope 作用域无关。

```
<div ng-controller="TestCtrl" ng-init="a=[1,2,3,4];">
  <ul ng-repeat="member in a">
    <li>{{ member }}</li>
  </ul>
</div>
```

7.3. 节点控制

7.3.1. 样式 ng-style

可以使用一个结构直接表示当前节点的样式：

```
<div ng-style="{width: 100 + 'px', height: 100 + 'px', backgroundColor: 'red'}">
</div>
```

同样地，绑定一个变量的话，威力大了。

7.3.2. 类 ng-class

就是直接地设置当前节点的类，同样，配合数据绑定作用就大了：

```
<div ng-controller="TestCtrl" ng-class="cls">
</div>
```

ng-class-even 和 *ng-class-odd* 是和 *ng-repeat* 配合使用的：

```
<ul ng-init="l=[1,2,3,4]">
  <li ng-class-odd="'odd'" ng-class-even="'even'" ng-repeat="m in l">{{ m }}</li>
</ul>
```

注意里面给的还是表示式，别少了引号。

7.3.3. 显示和隐藏 ng-show ng-hide ng-switch

前两个是控制 display 的指令：

```
<div ng-show="true">1</div>
<div ng-show="false">2</div>
<div ng-hide="true">3</div>
<div ng-hide="false">4</div>
```

后一个 ng-switch 是根据一个值来决定哪个节点显示，其它节点移除：

```
<div ng-init="a=2">
  <ul ng-switch on="a">
```

```
    <li ng-switch-when="1">1</li>
    <li ng-switch-when="2">2</li>
    <li ng-switch-default>other</li>
  </ul>
</div>
```

7.3.4. 其它属性控制

ng-src 控制 src 属性：

```
<img ng-src="{{ 'h' + 'ead.png' }}" />
```

ng-href 控制 href 属性：

```
<a ng-href="{{ '#' + '123' }}">here</a>
```

总的来说：

- ng-src src属性
- ng-href href属性
- ng-checked 选中状态
- ng-selected 被选择状态
- ng-disabled 禁用状态
- ng-multiple 多选状态
- ng-readonly 只读状态

注意： 上面的这些只是单向绑定，即只是从数据到展示，不能反作用于数据。要双向绑定，还是要使用 ng-model 。

7.4. 事件绑定

事件绑定是模板指令中很好用的一部分。我们可以把相关事件的处理函数直接写在 DOM 中，这样做的最大好处就是可以从 DOM 结构上看出业务处理的形式，你知道当你点击这个节点时哪个函数被执行了。

- ng-change

- ng-click
- ng-dblclick
- ng-mousedown
- ng-mouseenter
- ng-mouseleave
- ng-mousemove
- ng-mouseover
- ng-mouseup
- ng-submit

对于事件对象本身，在函数调用时可以直接使用 $event 进行传递：

```
<p ng-click="click($event)">点击</p>
<p ng-click="click($event.target)">点击</p>
```

7.5. 表单控件

表单控件类的模板指令，最大的作用是它预定义了需要绑定的数据的格式。这样，就可以对于既定的数据进行既定的处理。

7.5.1. form

form 是核心的一个控件。 ng 对 form 这个标签作了包装。事实上， ng 自己的指令是叫 ng-form 的，区别在于， form 标签不能嵌套，而使用 ng-form 指令就可以做嵌套的表单了。

form 的行为中依赖它里面的各个输入控制的状态的，在这里，我们主要关心的是 form 自己的一些方法和属性。从 ng 的角度来说， form 标签，是一个模板指令，也创建了一个 FormController 的实例。这个实例就提供了相应的属性和方法。同时，它里面的控件也是一个 NgModelController 实例。

很重要的一点， form 的相关方法要生效，必须为 form 标签指定 name 和 ng-controller ，并且每个控件都要绑定一个变量。 form 和控件的名字，即是 $scope 中的相关实例的引用变量名。

```
<form name="test_form" ng-controller="TestCtrl">
  <input type="text" name="a" required ng-model="a"  />
  <span ng-click="see()">{{ test_form.$valid }}</span>
</form>

var TestCtrl = function($scope){

  $scope.see = function(){
    console.log($scope.test_form);
    console.log($scope.test_form.a);
  }
```

除去对象的方法与属性， form 这个标签本身有一些动态类可以使用：

- ng-valid 当表单验证通过时的设置
- ng-invalid 当表单验证失败时的设置
- ng-pristine 表单的未被动之前拥有
- ng-dirty 表单被动过之后拥有

form 对象的属性有：

- $pristine 表单是否未被动过
- $dirty 表单是否被动过
- $valid 表单是否验证通过
- $invalid 表单是否验证失败
- $error 表单的验证错误

其中的 $error 对象包含有所有字段的验证信息，及对相关字段的 NgModelController 实例的引用。它的结构是一个对象， key 是失败信息， required ， minlength 之类的， value 是对应的字段实例列表。

注意，这里的失败信息是按序列取的一个。比如，如果一个字段既要求 required ，也要求 minlength ，那么当它为空时， $error 中只有 required 的失败信息。只输入一个字符之后， required 条件满足了，才可能有 minlength 这个失败信息。

```
<form name="test_form" ng-controller="TestCtrl">
  <input type="text" name="a" required ng-model="a"  />
  <input type="text" name="b" required ng-model="b" ng-minlength="2" />
  <span ng-click="see()">{{ test_form.$error }}</span>
</form>
```

```
var TestCtrl = function($scope){
  $scope.see = function(){
    console.log($scope.test_form.$error);
  }
}
```

7.5.2. input

input 是数据的最主要入口。 ng 支持 HTML5 中的相关属性，同时对旧浏览器也做了兼容性处理。最重要的， input 的规则定义，是所属表单的相关行为的参照（比如表单是否验证成功）。

input 控件的相关可用属性为：

- name 名字
- ng-model 绑定的数据

- required 是否必填
- ng-required 是否必填
- ng-minlength 最小长度
- ng-maxlength 最大长度
- ng-pattern 匹配模式
- ng-change 值变化时的回调

```
<form name="test_form" ng-controller="TestCtrl">
  <input type="text" name="a" ng-model="a" required ng-pattern="-/abc/" />
  <span ng-click="see()">{{ test_form.$error }}</span>
</form>
```

input 控件，它还有一些扩展，这些扩展有些有自己的属性：

- input type="number" 多了 number 错误类型，多了 max ， min 属性。
- input type="url" 多了 url 错误类型。
- input type="email" 多了 email 错误类型。

7.5.3. checkbox

它也算是 input 的扩展，不过，它没有验证相关的东西，只有选中与不选中两个值：

```
<form name="test_form" ng-controller="TestCtrl">
  <input type="checkbox" name="a" ng-model="a" ng-true-value="AA" ng-false-value="BB" />
  <span>{{ a }}</span>
</form>
```

```
var TestCtrl = function($scope){
  $scope.a = 'AA';
}
```

两点：

1. controller 要初始化变量值。
2. controller 中的初始化值会关系到控件状态（双向绑定）。

7.5.4. radio

也是 input 的扩展。和 checkbox 一样，但它只有一个值了：

```
<form name="test_form" ng-controller="TestCtrl">
  <input type="radio" name="a" ng-model="a" value="AA" />
  <input type="radio" name="a" ng-model="a" value="BB" />
  <span>{{ a }}</span>
</form>
```

7.5.5. textarea

同 input 。

7.5.6. select

这是一个比较牛B的控件。它里面的一个叫做 ng-options 的属性用于数据呈现。

对于给定列表时的使用。

最简单的使用方法， x for x in list：

```
<form name="test_form" ng-controller="TestCtrl" ng-init="o=[0,1,2,3]; a=o[1];">
  <select ng-model="a" ng-options="x for x in o" ng-change="show()">
    <option value="">可以加这个空值</option>
  </select>
```

```
</form>

<script type="text/javascript">
var TestCtrl = function($scope){
  $scope.show = function(){
    console.log($scope.a);
  }
}

angular.bootstrap(document.documentElement);
</script>
```

在 $scope 中， select 绑定的变量，其值和普通的 value 无关，可以是一个对象：

```
<form name="test_form" ng-controller="TestCtrl"
    ng-init="o=[{name: 'AA'}, {name: 'BB'}]; a=o[1];">
  <select ng-model="a" ng-options="x.name for x in o" ng-change="show()">
  </select>
</form>
```

显示与值分别指定， x.v as x.name for x in o：

```
<form name="test_form" ng-controller="TestCtrl"
    ng-init="o=[{name: 'AA', v: '00'}, {name: 'BB', v: '11'}]; a=o[1].v;">
  <select ng-model="a" ng-options="x.v as x.name for x in o" ng-change="show()">
  </select>
```

```
</form>
```

加入分组的， x.name group by x.g for x in o：

```
<form name="test_form" ng-controller="TestCtrl"
    ng-init="o=[{name: 'AA', g: '00'}, {name: 'BB', g: '11'}, {name: 'CC', g: '00'}]; a=o[1];">
    <select ng-model="a" ng-options="x.name group by x.g for x in o" ng-change="show()">
    </select>
</form>
```

分组了还分别指定显示与值的， x.v as x.name group by x.g for x in o：

```
<form name="test_form" ng-controller="TestCtrl" ng-init="o=[{name: 'AA', g: '00', v: '='}, {name: 'BB', g: '11', v: '+'}, {name: 'CC', g: '00', v: '!'}]; a=o[1].v;">
    <select ng-model="a" ng-options="x.v as x.name group by x.g for x in o" ng-change="show()">
    </select>
</form>
```

如果参数是对象的话，基本也是一样的，只是把遍历的对象改成 (key, value)：

```
<form name="test_form" ng-controller="TestCtrl" ng-init="o={a: 0, b: 1}; a=o.a;">
    <select ng-model="a" ng-options="k for (k, v) in o" ng-change="show()">
    </select>
</form>
```

```
<form name="test_form" ng-controller="TestCtrl"
    ng-init="o={a: {name: 'AA', v: '00'}, b: {name: 'BB', v: '11'}};
a=o.a.v;">
  <select ng-model="a" ng-options="v.v as v.name for (k, v) in o" ng-
change="show()">
  </select>
</form>

<form name="test_form" ng-controller="TestCtrl"
    ng-init="o={a: {name: 'AA', v: '00', g: '=='}, b: {name: 'BB', v: '11', g:
'=='}}; a=o.a;">
  <select ng-model="a" ng-options="v.name group by v.g for (k, v) in o"
ng-change="show()">
  </select>
</form>

<form name="test_form" ng-controller="TestCtrl"
    ng-init="o={a: {name: 'AA', v: '00', g: '=='}, b: {name: 'BB', v: '11', g:
'=='}}; a=o.a.v;">
  <select ng-model="a" ng-options="v.v as v.name group by v.g for (k,
v) in o" ng-change="show()">
  </select>
</form>
```

8. 模板中的过滤器

这里说的过滤器，是用于对数据的格式化，或者筛选的函数。它们可以直接在模板中通过一种语法使用。对于常用功能来说，是很方便的一种机制。

多个过滤器之间可以直接连续使用。

8.1. 排序 orderBy

orderBy 是一个排序用的过滤器标签。它可以像 sort 函数那样支持一个排序函数，也可以简单地指定一个属性名进行操作：

```
<div ng-controller="TestCtrl">
  {{ data | orderBy: 'age' }} <br />
  {{ data | orderBy: '-age' }} <br />
  {{ data | orderBy: '-age' | limitTo: 2 }} <br />
  {{ data | orderBy: ['-age', 'name'] }} <br />
</div>


<script type="text/javascript">
var TestCtrl = function($scope){
  $scope.data = [
    {name: 'B', age: 4},
    {name: 'A', age: 1},
    {name: 'D', age: 3},
    {name: 'C', age: 3},
  ];
}
```

angular.bootstrap(document.documentElement);

</script>

8.2. 过滤列表 filter

filter 是一个过滤内容的标签。

如果参数是一个字符串，则列表成员中的任意属性值中有这个字符串，即为满足条件（忽略大小写）：

```
<div ng-controller="TestCtrl">
  {{ data | filter: 'b' }} <br />
  {{ data | filter: '!B' }} <br />
</div>
```

```
<script type="text/javascript">
var TestCtrl = function($scope){
  $scope.data = [
    {name: 'B', age: 4},
    {name: 'A', age: 1},
    {name: 'D', age: 3},
    {name: 'C', age: 3},
  ];
}

angular.bootstrap(document.documentElement);
```

```
  </script>
```

可以使用对象，来指定属性名， $ 表示任意属性：

```
  {{ data | filter: {name: 'A'} }} <br />
  {{ data | filter: {$: '3'} }} <br />
  {{ data | filter: {$: '!3'} }} <br />
```

自定义的过滤函数也支持：

```
  <div ng-controller="TestCtrl">
    {{ data | filter: f }} <br />
  </div>



  <script type="text/javascript">
var TestCtrl = function($scope){
  $scope.data = [
    {name: 'B', age: 4},
    {name: 'A', age: 1},
    {name: 'D', age: 3},
    {name: 'C', age: 3},
  ];

  $scope.f = function(e){
    return e.age > 2;
  }
}
```

angular.bootstrap(document.documentElement);

</script>

8.3. 其它

时间戳格式化 date：

* *

{{ a | date: 'yyyy-MM-dd HH:mm:ss' }}

* *

<script type="text/javascript">

var TestCtrl = function($scope){

$scope.a = ((new Date().valueOf()));

}

angular.bootstrap(document.documentElement);

</script>

列表截取 limitTo，支持正负数:

{{ [1,2,3,4,5] | limitTo: 2 }}

{{ [1,2,3,4,5] | limitTo: -3 }}

大小写 lowercase， uppercase：

{{ 'abc' | uppercase }}

{{ 'Abc' | lowercase }}

8.4. 例子：表头排序

```
<div ng-controller="TestCtrl">
  <table>
    <tr>
      <th ng-click="f='name'; rev=!rev">名字</th>
      <th ng-click="f='age'; rev=!rev">年龄</th>
    </tr>

    <tr ng-repeat="o in data | orderBy: f : rev">
      <td>{{ o.name }}</td>
      <td>{{ o.age }}</td>
    </tr>
  </table>
</div>

<script type="text/javascript">
var TestCtrl = function($scope){
  $scope.data = [
    {name: 'B', age: 4},
    {name: 'A', age: 1},
    {name: 'D', age: 3},
    {name: 'C', age: 3},
  ];
}

angular.bootstrap(document.documentElement);
</script>
```

8.5. 例子：搜索

<div ng-controller="TestCtrl" ng-init="s=data[0].name; q="">

* *

查找： <input type="text" ng-model="q" />

* *

<select ng-multiple="true" ng-model="s"

ng-options="o.name as o.name + '(' + o.age + ')' for o in data | filter: {name: q} | orderBy: ['age', 'name'] ">

</select>

```
</div>

<script type="text/javascript">
var TestCtrl = function($scope){
  $scope.data = [
    {name: 'B', age: 4},
    {name: 'A', age: 1},
    {name: 'D', age: 3},
    {name: 'C', age: 3},
  ];
}

  angular.bootstrap(document.documentElement);
  </script>
```

9. 锚点路由

准确地说，这应该叫对 hashchange 事件的处理吧。

就是指 URL 中的锚点部分发生变化时，触发预先定义的业务逻辑。比如现在是 /test#/x ，锚点部分的值为 # 后的 /x ，它就对应了一组处理逻辑。当这部分变化时，比如变成了 /test#/t ，这时页面是不会刷新的，但是它可以触发另外一组处理逻辑，来做一些事，也可以让页面发生变化。

这种机制对于复杂的单页面来说，无疑是一种强大的业务切分手段。就算不是复杂的单页面应用，在普通页面上善用这种机制，也可以让业务逻辑更容易控制。

ng 提供了完善的锚点路由功能，虽然目前我觉得相当重要的一个功能还有待完善（后面会说），但目前这功能的几部分内容，已经让我思考了很多种可能性了。

ng 中的锚点路由功能是由几部分 API 共同完成的一整套方案。这其中包括了路由定义，参数定义，业务处理等。

9.1. 路由定义

要使用锚点路由功能，需要在先定义它。目前，对于定义的方法，我个人只发现在“初始化”阶段可以通过 $routeProvider 这个服务来定义。

在定义一个 app 时可以定义锚点路由：

```
<html ng-app="ngView">

  ... ...


<div ng-view></div>

<script type="text/javascript">

angular.module('ngView', [],
  function($routeProvider){
    $routeProvider.when('/test',
      {
        template: 'test',
      }
    );
  }
);
```

```
</script>
```

首先看 ng-view 这个 directive ，它是一个标记“锚点作用区”的指令。目前页面上只能有一个“锚点作用区”。有人已经提了，“多个可命名”的锚点作用区的代码到官方，但是目前官方还没有接受合并，我觉得多个作用区这个功能是很重要的，希望下个发布版中能有。

锚点作用区的功能，就是让锚点路由定义时的那些模板， controller 等，它们产生的 HTML 代码放在作用区内。

比如上面的代码，当你刚打开页面时，页面是空白的。你手动访问 /#/test 就可以看到页面上出现了 'test' 的字样。

在 angular.bootstrap() 时也可以定义：

```
angular.bootstrap(document.documentElement, [
  function($routeProvider){
    $routeProvider.when('/test',
      {
        template: 'test'
      }
    );
  }
]);
```

9.2. 参数定义

在作路由定义时，可以匹配一个规则，规则中可以定义路径中的某些部分作为参数之用，然后使用 $routeParams 服务获取到指定参数。比如 /#/book/test 中， test 作为参数传入到 controller 中：

```
<div ng-view></div>
```

```
<script type="text/javascript">

angular.module('ngView', [],
  function($routeProvider){
    $routeProvider.when('/book/:title',
      {
        template: '{{ title }}',
        controller: function($scope, $routeParams){
          $scope.title = $routeParams.title;
        }
      }
    );
  }
);

 </script>
```

访问： /#/book/test

不需要预定义模式，也可以像普通 GET 请求那样获取到相关参数：

```
 angular.module('ngView', [],
   function($routeProvider){
     $routeProvider.when('/book',
       {
         template: '{{ title }}',
```

```
        controller: function($scope, $routeParams){
          $scope.title = $routeParams.title;
        }
      }
    );
  }
);
```

访问： /#/book?title=test

9.3. 业务处理

简单来说，当一个锚点路由定义被匹配时，会根据模板生成一个 $scope ，同时相应的一个 controller 就会被触发。最后模板的结果会被填充到 ng-view 中去。

从上面的例子中可以看到，最直接的方式，我们可以在模板中双向绑定数据，而数据的来源，在 controller 中控制。在 controller 中，又可以使用到像 $scope ， $routeParams 这些服务。

这里先提一下另外一种与锚点路由相关的服务， $route 。这个服务里锚点路由在定义时，及匹配过程中的信息。比如我们搞怪一下：

```
angular.module('ngView', [],
  function($routeProvider){
    $routeProvider.when('/a',
      {
        template: '{{ title }}',
        controller: function($scope){
          $scope.title = 'a';
        }
```

```
  }
);

$routeProvider.when('/b',
  {
    template: '{{ title }}',
    controller: function($scope, $route){
      console.log($route);
      $route.routes['/a'].controller($scope);
    }
  }
);
```

回到锚点定义的业务处理中来。我们可以以字符串形式写模板，也可以直接引用外部文件作为模板：

```
angular.module('ngView', [],
  function($routeProvider){
    $routeProvider.when('/test',
      {
        templateUrl: 'tpl.html',
        controller: function($scope){
          $scope.title = 'a';
        }
```

```
        }
      );
    }
  );
```

tpl.html 中的内容是：

```
{{ title }}
```

这样的话，模板可以预定义，也可以很复杂了。

现在暂时忘了模板吧，因为前面提到的，当前 ng-view 不能有多个的限制，模板的渲染机制局限性还是很大的。不过，反正会触发一个 controller，那么在函数当中我们可以尽量地干自己喜欢的事：

```
angular.module('ngView', [],
  function($routeProvider){
    $routeProvider.when('/test',
      {
        template: '{{}}',
        controller: function(){
          $('div').first().html('<b>OK</b>');
        }
      }
    );
  }
);
```

那个空的 template 不能省，否则 controller 不会被触发。

10. 定义模板变量标识标签

由于下面涉及动态内容，所以我打算起一个后端服务来做。但是我发现我使用的 Tornado 框架的模板系统，与 ng 的模板系统，都是使用 {{ }} 这对符号来定义模板表达式的，这太悲剧了，不过幸好 ng 已经提供了修改方法：

```
angular.bootstrap(document.documentElement,
  [function($interpolateProvider){
    $interpolateProvider.startSymbol('[[');
    $interpolateProvider.endSymbol(']]');
  }]);
```

使用 $interpolateProvider 服务即可。

11. AJAX

ng 提供了基本的 AJAX 封装，你直接面对 promise 对象，使用起来还是很方便的。

11.1. HTTP请求

基本的操作由 $http 服务提供。它的使用很简单，提供一些描述请求的参数，请求就出去了，然后返回一个扩充了 success 方法和 error 方法的 promise 对象（下节介绍），你可以在这个对象中添加需要的回调函数。

```
var TestCtrl = function($scope, $http){
  var p = $http({
    method: 'GET',
    url: '/json'
  });
  p.success(function(response, status, headers, config){
```

```
        $scope.name = response.name;
    });
}
```

$http 接受的配置项有：

- method 方法
- url 路径
- params GET请求的参数
- data post请求的参数
- headers 头
- transformRequest 请求预处理函数
- transformResponse 响应预处理函数
- cache 缓存
- timeout 超时毫秒，超时的请求会被取消
- withCredentials 跨域安全策略的一个东西

其中的 transformRequest 和 transformResponse 及 headers 已经有定义的，如果自定义则会覆盖默认定义：

```
var $config = this.defaults = {
  // transform incoming response data
  transformResponse: [function(data) {
    if (isString(data)) {
      // strip json vulnerability protection prefix
      data = data.replace(PROTECTION_PREFIX, '');
      if (JSON_START.test(data) && JSON_END.test(data))
        data = fromJson(data, true);
    }
    return data;
  }],

  // transform outgoing request data
  transformRequest: [function(d) {
    return isObject(d) && !isFile(d) ? toJson(d) : d;
  }],
```

```
17
18      // default headers
19      headers: {
20        common: {
21          'Accept': 'application/json, text/plain, */*',
22          'X-Requested-With': 'XMLHttpRequest'
23        },
24        post: {'Content-Type': 'application/json;charset=utf-8'},
25        put:  {'Content-Type': 'application/json;charset=utf-8'}
26      }
27    };
```

注意它默认的 POST 方法出去的 Content-Type

对于几个标准的 HTTP 方法，有对应的 shortcut：

- $http.delete(url, config)
- $http.get(url, config)
- $http.head(url, config)
- $http.jsonp(url, config)
- $http.post(url, data, config)
- $http.put(url, data, config)

注意其中的 JSONP 方法，在实现上会在页面中添加一个 script 标签，然后放出一个 GET 请求。你自己定义的，匿名回调函数，会被 ng 自己给一个全局变量。在定义请求，作为 GET 参数，你可以使用 JSON_CALLBACK 这个字符串来暂时代替回调函数名，之后 ng 会为你替换成真正的函数名：

```
var p = $http({
  method: 'JSONP',
  url: '/json',
  params: {callback: 'JSON_CALLBACK'}
});
```

```
p.success(function(response, status, headers, config){
    console.log(response);
    $scope.name = response.name;
});
```

$http 有两个属性：

- defaults 请求的全局配置
- pendingRequests 当前的请求队列状态

```
$http.defaults.transformRequest = function(data){console.log('here');
return data;}
console.log($http.pendingRequests);
```

11.2. 广义回调管理

和其它框架一样， ng 提供了广义的异步回调管理的机制。 $http 服务是在其之上封装出来的。这个机制就是 ng 的 $q 服务。

不过 ng 的这套机制总的来说实现得比较简单，按官方的说法，够用了。

使用的方法，基本上是：

- 通过 $q 服务得到一个 deferred 实例
- 通过 deferred 实例的 promise 属性得到一个 promise 对象
- promise 对象负责定义回调函数
- deferred 实例负责触发回调

```
var TestCtrl = function($q){
  var defer = $q.defer();
  var promise = defer.promise;
  promise.then(function(data){console.log('ok, ' + data)},
               function(data){console.log('error, ' + data)});
```

```
    //defer.reject('xx');
    defer.resolve('xx');
  }
```

了解了上面的东西，再分别看 $q ， deferred ， promise 这三个东西。

11.2.1. $q

$q 有四个方法：

- $q.all() 合并多个 promise ，得到一个新的 promise
- $q.defer() 返回一个 deferred 对象
- $q.reject() 包装一个错误，以使回调链能正确处理下去
- $q.when() 返回一个 promise 对象

$q.all() 方法适用于并发场景很合适：

```
  var TestCtrl = function($q, $http){
    var p = $http.get('/json', {params: {a: 1}});
    var p2 = $http.get('/json', {params: {a: 2}});
    var all = $q.all([p, p2]);
    p.success(function(res){console.log('here')});
    all.then(function(res){console.log(res[0])});
  }
```

$q.reject() 方法是在你捕捉异常之后，又要把这个异常在回调链中传下去时使用：

要理解这东西，先看看 promise 的链式回调是如何运作的，看下面两段代码的区别：

```
    var defer = $q.defer();
    var p = defer.promise;
    p.then(
```

```
  function(data){return 'xxx'}
);
p.then(
  function(data){console.log(data)}
);
defer.resolve('123');
var defer = $q.defer();
var p = defer.promise;
var p2 = p.then(
  function(data){return 'xxx'}
);
p2.then(
  function(data){console.log(data)}
);
defer.resolve('123');
```

从模型上看，前者是“并发”，后者才是“链式”。

而 $q.reject() 的作用就是触发后链的 error 回调：

```
var defer = $q.defer();
var p = defer.promise;
p.then(
  function(data){return data},
  function(data){return $q.reject(data)}
).
then(
```

```
    function(data){console.log('ok, ' + data)},
    function(data){console.log('error, ' + data)}
  )
  defer.reject('123');
```

最后的 $q.when() 是把数据封装成 promise 对象：

```
  var p = $q.when(0, function(data){return data},
                     function(data){return data});
  p.then(
    function(data){console.log('ok, ' + data)},
    function(data){console.log('error, ' + data)}
  );
```

11.2.2. deferred

deferred 对象有两个方法一个属性。

- promise 属性就是返回一个 promise 对象的。
- resolve() 成功回调
- reject() 失败回调

```
  var defer = $q.defer();
  var promise = defer.promise;
  promise.then(function(data){console.log('ok, ' + data)},
               function(data){console.log('error, ' + data)});
  //defer.reject('xx');
  defer.resolve('xx');
```

11.2.3. promise

promise 对象只有 then() 一个方法，注册成功回调函数和失败回调函数，再返回一个 promise 对象，以用于链式调用。

12. 工具函数

12.1. 上下文绑定

angular.bind 是用来进行上下文绑定，参数动态绑定的工具函数。

```
var f = angular.bind({a: 'xx'},
  function(){
    console.log(this.a);
  }
);
f();
```

参数动态绑定：

```
var f = function(x){console.log(x)}
angular.bind({}, f, 'x')();
```

12.2. 对象处理

对象复制： angular.copy()

```
var a = {'x': '123'};
var b = angular.copy(a);
a.x = '456';
console.log(b);
```

对象聚合： *angular.extend()*

```
var a = {'x': '123'};
var b = {'xx': '456'};
```

```
  angular.extend(b, a);
  console.log(b);
```

空函数： angular.noop()

大小写转换： angular.lowercase() 和 angular.uppercase()

JSON转换： angular.fromJson() 和 angular.toJson()

遍历： angular.forEach()，支持列表和对象：

```
  var l = {a: '1', b: '2'};
  angular.forEach(l, function(v, k){console.log(k + ': ' + v)});

  var l = ['a', 'b', 'c'];
  angular.forEach(l, function(v, i, o){console.log(v)});

  var context = {'t': 'xx'};
  angular.forEach(l, function(v, i, o){console.log(this.t)}, context);
```

12.3. 类型判定

- angular.isArray
- angular.isDate
- angular.isDefined
- angular.isElement
- angular.isFunction
- angular.isNumber
- angular.isObject
- angular.isString
- angular.isUndefined

13. 其它服务

13.1. 日志

ng 提供 $log 这个服务用于向终端输出相关信息：

- *error()*
- *info()*
- *log()*
- *warn()*

```
var TestCtrl = function($log){
  $log.error('error');
  $log.info('info');
  $log.log('log');
  $log.warn('warn');
}
```

13.2. 缓存

ng 提供了一个简单封装了缓存机制 $cacheFactory ， 可以用来作为数据容器：

```
var TestCtrl = function($scope, $cacheFactory){
  $scope.cache = $cacheFactory('s_' + $scope.$id, {capacity: 3});

  $scope.show = function(){
    console.log($scope.cache.get('a'));
    console.log($scope.cache.info());
  }
```

```
  $scope.set = function(){
    $scope.cache.put((new Date()).valueOf(), 'ok');
  }
}
```

调用时，第一个参数是 id ，第二个参数是配置项，目前支持 capacity 参数，用以设置缓存能容留的最大条目数。超过这个个数，则自动清除较旧的条目。

缓存实例的方法：

- info() 获取 id , size 信息
- put(k, v) 设置新条目
- get(k) 获取条目
- remove(k) 删除条目
- removeAll() 删除所有条目
- destroy() 删除对本实例的引用

$http 的调用当中，有一个 cache 参数，值为 true 时为自动维护的缓存。值也可以设置为一个 cache 实例。

13.3. 计时器

$timeout 服务是 ng 对 window.setTimeout() 的封装，它使用 promise 统一了计时器的回调行为：

```
var TestCtrl = function($timeout){
  var p = $timeout(function(){console.log('haha')}, 5000);
  p.then(function(){console.log('x')});
  //$timeout.cancel(p);
}
```

使用 $timeout.cancel() 可以取消计时器。

13.4. 表达式函数化

$parse 这个服务，为 js 提供了类似于 Python 中 @property 的能力：

```
var TestCtrl = function($scope, $parse){
  $scope.get_name = $parse('name');
  $scope.show = function(){console.log($scope.get_name($scope))}
  $scope.set = function(){$scope.name = '123'}
}
```

$parse 返回一个函数，调用这个函数时，可以传两个参数，第一个作用域，第二个是变量集，后者常用于覆盖前者的变量：

```
var get_name = $parse('name');
var r = get_name({name: 'xx'}, {name: 'abc'});
console.log(r);
```

$parse 返回的函数，也提供了相应的 assign 功能，可以为表达式赋值（如果可以的话）：

```
var get_name = $parse('name');
var set_name = get_name.assign;
var r = get_name({name: 'xx'}, {name: 'abc'});
console.log(r);

var s = {}
set_name(s, '123');
var r = get_name(s);
console.log(r);
```

13.5. 模板单独使用

ng 中的模板是很重要，也很强大的一个机制，自然少不了单独运用它的方法。不过，即使是单独使用，也是和 DOM 紧密相关的程度：

- 定义时必须是有 HTML 标签包裹的，这样才能创建 DOM 节点
- 渲染时必须传入 $scope

之后使用 $compile 就可以得到一个渲染好的节点对象了。当然， $compile 还要做其它一些工作，指令处理什么的。

```
var TestCtrl = function($scope, $element,$compile){
  $scope.a = '123';
  $scope.set = function(){
    var tpl = $compile('<p>hello {{ a }}</p>');
    var e = tpl($scope);
    $element.append(e);
  }
}
```

14. 自定义模块和服务

14.1. 模块和服务的概念与关系

总的来说，模块是组织业务的一个框框，在一个模块当中定义多个服务。当你引入了一个模块的时候，就可以使用这个模块提供的一种或多种服务了。

比如 AngularJS 本身的一个默认模块叫做 ng ， 它提供了 $http ， $q 等等服务。

服务只是模块提供的多种机制中的一种，其它的还有命令（ directive ），过滤器（ filter ），及其它配置信息。

然后在额外的 js 文件中有一个附加的模块叫做 ngResource， 它提供了一个 $resource 服务。

定义时，我们可以在已有的模块中新定义一个服务，也可以先新定义一个模块，然后在新模块中定义新服务。

使用时，模块是需要显式地的声明依赖（引入）关系的，而服务则可以让 ng 自动地做注入，然后直接使用。

14.2. 定义模块

定义模块的方法是使用 angular.module 。调用时声明了对其它模块的依赖，并定义了“初始化”函数。

```
var my_module = angular.module('MyModule', [], function(){
    console.log('here');
});
```

这段代码定义了一个叫做 MyModule 的模块， my_module 这个引用可以在接下来做其它的一些事，比如定义服务。

14.3. 定义服务

服务本身是一个任意的对象。但是 ng 提供服务的过程涉及它的依赖注入机制。在这里呢，就要先介绍一下叫 provider 的东西。

简单来说， provider 是被“注入控制器”使用的一个对象，注入机制通过调用一个 provider 的 $get() 方法，把得到的东西作为参数进行相关调用（比如把得到的服务作为一个 Controller 的参数）。

在这里“服务”的概念就比较不明确，对使用而言，服务仅指 $get() 方法返回的东西，但是在整体机制上，服务又要指提供了 $get() 方法的整个对象。

```
//这是一个provider
var pp = function(){
  this.$get = function(){
    return {'haha': '123'};
```

```
  }
}

//我在模块的初始化过程当中, 定义了一个叫 PP 的服务
var app = angular.module('Demo', [], function($provide){
  $provide.provider('PP', pp);
});

//PP服务实际上就是 pp 这个 provider 的 $get() 方法返回的东西
app.controller('TestCtrl',
  function($scope, PP){
    console.log(PP);
  }
);
```

上面的代码是一种定义服务的方法，当然， ng 还有相关的 shortcut，ng 总有很多 shortcut 。

第一个是 factory 方法，由 $provide 提供， module 的 factory 是一个引用，作用一样。这个方法直接把一个函数当成是一个对象的 $get() 方法，这样你就不用显式地定义一个 provider 了：

```
var app = angular.module('Demo', [], function($provide){
  $provide.factory('PP', function(){
    return {'hello': '123'};
  });
});
app.controller('TestCtrl', function($scope, PP){ console.log(PP) });
```

在 module 中使用：

```
var app = angular.module('Demo', [], function(){ });
app.factory('PP', function(){return {'abc': '123'}});
app.controller('TestCtrl', function($scope, PP){ console.log(PP) });
```

第二个是 service 方法，也是由 $provide 提供， module 中有对它的同名引用。 service 和 factory 的区别在于，前者是要求提供一个“构造方法”，后者是要求提供 $get() 方法。意思就是，前者一定是得到一个 object ，后者可以是一个数字或字符串。它们的关系大概是：

```
var app = angular.module('Demo', [], function(){ });
app.service = function(name, constructor){
  app.factory(name, function(){
    return (new constructor());
  });
}
```

这里插一句，js 中 new 的作用，以 new a() 为例，过程相当于：

1. 创建一个空对象 obj
2. 把 obj 绑定到 a 函数的上下文当中（即 a 中的 this 现在指向 obj ）
3. 执行 a 函数
4. 返回 obj

service 方法的使用就很简单了：

```
var app = angular.module('Demo', [], function(){ });
app.service('PP', function(){
  this.abc = '123';
});
app.controller('TestCtrl', function($scope, PP){ console.log(PP) });
```

14.4. 引入模块并使用服务

结合上面的“定义模块”和“定义服务”，我们可以方便地组织自己的额外代码：

```
angular.module('MyModule', [], function($provide){
  $provide.factory('S1', function(){
    return 'I am S1';
  });
  $provide.factory('S2', function(){
    return {see: function(){return 'I am S2'}}
  });
});

var app = angular.module('Demo', ['MyModule'], angular.noop);
app.controller('TestCtrl', function($scope, S1, S2){
  console.log(S1)
  console.log(S2.see())
});
```

15. 附加模块 ngResource

15.1. 使用引入与整体概念

ngResource 这个是 ng 官方提供的一个附加模块。附加的意思就是，如果你打算用它，那么你需要引入一人单独的 js 文件，然后在声明“根模块”时注明依赖的 ngResource 模块，接着就可以使用它提供的 $resource 服务了。完整的过程形如：

```
<!DOCTYPE html>
```

```
<html ng-app="Demo">
<head>
<meta charset="utf-8" />
<title>AngularJS</title>
<script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.3/angular.min.js"></script>
<script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.3/angular-resource.js"></script>
</head>
<body>
  <div ng-controller="TestCtrl"></div>


<script type="text/javascript" charset="utf-8">

var app = angular.module('Demo', ['ngResource'], angular.noop);
app.controller('TestCtrl', function($scope, $resource){
  console.log($resource);
});

</script>
</body>
```

```
</html>
```

$resource 服务，整体上来说，比较像是使用类似 ORM 的方式来包装了 AJAX 调用。区别就是 ORM 是操作数据库，即拼出 SQL 语句之后，作 execute 方法调用。而 $resource 的方式是构造出 AJAX 请求，然后发出请求。同时，AJAX 请求是需要回调处理的，这方面， $resource 的机制可以使你在一些时候省掉回调处理，当然，是否作回调处理在于业务情形及容错需求了。

使用上 $resource 分成了“类”与“实例”这两个层面。一般地，类的方法调用就是直观的调用形式，通常会返回一个对象，这个对象即为“实例”。

“实例”贯穿整个服务的使用过程。“实例”的数据是填充方式，即因为异步关系，回调函数没有执行时，实例已经存在，只是可能它还没有相关数据，回调执行之后，相关数据被填充到实例对象当中。实例的方法一般就是在类方法名前加一个 $ ，调用上，根据定义，实例数据可能会做一些自动的参数填充，这点是区别实例与类的调用上的不同。

好吧，上面这些话可能需要在看了接下来的内容之后再回过来理解。

15.2. 基本定义

就像使用 ORM 一般要先定义 Model 一样，使用 $resource 需要先定义“资源”，也就是先定义一些 HTTP 请求。

在业务场景上，我们假设为，我们需要操作“书”这个实体，包括创建create，获取详情read，修改update，删除delete，批量获取multi，共五个操作方法。实体属性有：唯一标识id，标题title，作者author。

我们把这些操作定义成 $resource 的资源：

```
var app = angular.module('Demo', ['ngResource'], angular.noop);
app.controller('BookCtrl', function($scope, $resource){
  var actions = {
    create: {method: 'POST', params: {_method: 'create'}},
    read: {method: 'POST', params: {_method: 'read'}},
    update: {method: 'POST', params: {_method: 'update'}},
```

```
    delete: {method: 'POST', params: {_method: 'delete'}},
    multi: {method: 'POST', params: {_method: 'multi'}}
  }
  var Book = $resource('/book', {}, actions);
});
```

定义是使用使用 $resource 这个函数就可以了，它接受三个参数：

- url
- 默认的params（这里的 params 即是 GET 请求的参数，POST 的参数单独叫做“postData”）
- 方法映射

方法映射是以方法名为 key，以一个对象为 value，这个 value 可以有三个成员：

- method, 请求方法，'GET', 'POST', 'PUT', 'DELETE' 这些
- params, 默认的 GET 参数
- isArray, 返回的数据是不是一个列表

15.3. 基本使用

在定义了资源之后，我们看如果使用这些资源，发出请求：

```
var book = Book.read({id: '123'}, function(response){
  console.log(response);
});
```

这里我们进行 Book 的“类”方法调用。在方法的使用上，根据官方文档：

HTTP GET "class" actions: Resource.action([parameters], [success], [error])

non-GET "class" actions: Resource.action([parameters], postData, [success], [error])

non-GET instance actions: instance.$action([parameters], [success], [error])

我们这里是第二种形式，即类方法的非 GET 请求。我们给的参数会作为 postData 传递。如果我们需要 GET 参数，并且还需要一个错误回调，那么：

```
var book = Book.read({get: 'haha'}, {id: '123'},
  function(response){
    console.log(response);
  },
  function(error){
    console.log(error);
  }
);
```

调用之后，我们会立即得到的 book ，它是 Book 类的一个实例。这里所谓的实例，实际上就是先把所有的 action 加一个 $ 前缀放到一个空对象里，然后把发出的参数填充进去。等请求返回了，把除 action 以外的成员删除掉，再把请求返回的数据填充到这个对象当中。所以，如果我们这样：

```
var book = Book.read({id: '123'}, function(response){
  console.log(book);
});
console.log(book)
```

就能看到 book 实例的变化过程了。

现在我们得到一个真实的实例，看一下实例的调用过程：

```
//响应的数据是 {result: 0, msg: '', obj: {id: 'xxx'}}
var book = Book.create({title: '测试标题', author: '测试作者'}, function(response){
```

```
  console.log(book);
});
```

可以看到，在请求回调之后， book 这个实例的成员已经被响应内容填充了。但是这里有一个问题，我们返回的数据，并不适合一个 book 实例。格式先不说，它把 title 和 author 这些信息都丢了（因为响应只返回了 id ）。

如果仅仅是格式问题，我们可以通过配置 $http 服务来解决（ AJAX 请求都要使用 $http 服务的）：

```
$http.defaults.transformResponse = function(data){return angular.fromJson(data).obj};
```

当然，我们也可以自己来解决一下丢信息的问题：

```
var p = {title: '测试标题', author: '测试作者'};
var book = Book.create(p, function(response){
  angular.extend(book, p);
  console.log(book);
});
```

不过，始终会有一些不方便了。比较正统的方式应该是调节服务器端的响应，让服务器端也具有和前端一样的实例概念，返回的是完整的实例信息。即使这样，你也还要考虑格式的事。

现在我们得到了一个真实的 book 实例了，带有 id 信息。我们尝试一下实例的方法调用，先回过去头看一下那三种调用形式，对于实例只有第三种形式：

```
non-GET instance actions: instance.$action([parameters], [success], [error])
```

首先解决一个疑问，如果一个实例是进行一个 GET 的调用会怎么样？没有任何问题，这当然没有任何问题的，形式和上面一样。

如何实例是做 POST 请求的话，从形式上看，我们无法控制请求的 postData？是的，所有的 POST 请求，其 postData 都会被实例数据自动填充，形式上我们只能控制 params 。

所以，如果是在做修改调用的话：

```
book.$update({title: '新标题', author: '测试作者'}, function(response){
  console.log(book);
});
```

这样是没有意义的并且错误的。因为要修改的数据只是作为 GET 参数传递了，而 postData 传递的数据就是当前实例的数据，并没有任何修改。

正确的做法：

```
book.title = '新标题'
book.$update(function(response){
  console.log(book);
});
```

显然，这种情况下，回调都可以省了：

```
book.title = '新标题'
book.$update();
```

15.4. 定义和使用时的占位量

两方面。一是在定义时，在其 URL 中可以使用变量引用的形式（类型于定义锚点路由时那样）。第二时定义默认 params ，即 GET 参数时，可以定义为引用 postData 中的某变量。比如我们这样改一下：

```
var Book = $resource('/book/:id', {}, actions);
var book = Book.read({id: '123'}, {}, function(response){
  console.log(response);
});
```

在 URL 中有一个 :id ，表示对 params 中 id 这个变量的引用。因为 read 是一个 POST 请求，根据调用形式，第一个参数是 params ，第二个参数是 postData 。这样的调用结果就是，我们会发一个 POST 请求到如下地址， postData 为空：

```
/book/123?_method=read
```

再看默认的 params 中引用 postData 变量的形式：

```
var Book = $resource('/book', {id: '@id'}, actions);
var book = Book.read({title: 'xx'}, {id: '123'}, function(response){
  console.log(response);
});
```

这样会出一个 POST 请求， postData 内容中有一个 id 数据，访问的 URL 是：

```
/book?_method=read&id=123&title=xx
```

这两个机制也可以联合使用：

```
var Book = $resource('/book/:id', {id: '@id'}, actions);
var book = Book.read({title: 'xx'}, {id: '123'}, function(response){
  console.log(response);
});
```

结果就是出一个 POST 请求， postData 内容中有一个 id 数据，访问的 URL 是：

```
/book/123?_method=read&title=xx
```

15.5. 实例

ngResource 要举一个实例是比较麻烦的事。因为它必须要一个后端来支持，这里如果我用 Python 写一个简单的后端，估计要让这个后端跑起来对很多人来说都是问题。所以，我在几套公共服务的 API 中纠结考察了一番，最后使用 www.rememberthemilk.com 的 API 来做了一个简单的，可用的例子。

例子见： http://zouyesheng.com/demo/ng-resource-demo.html (可以直接下载看源码)

先说一下 API 的情况。这里的请求调用全是跨域的，所以交互上全部是使用了 JSONP 的形式。 API 的使用有使用签名认证机制，嗯， js 中直接算 md5 是可行的，我用了一个现成的库（但是好像不能处理中文吧）。

这个例子中的 LoginCtrl 大家就不用太关心了，参见官方的文档，走完流程拿到 token 完事。与 ngResource 相关的是 MainCtrl 中的东西。

其实从这个例子中就可以看出，目前 ngResource 的机制对于服务端返回的数据的格式是严重依赖的，同时也可以反映出 $http 对一些场景根本无法应对的局限。所以，我现在的想法是理解 ngResource 的思想，真正需要的人自己使用 jQuery 重新实现一遍也许更好。这应该也花不了多少时间，ngResource 的代码本来不多。

我为什么说 $http 在一些场景中有局限呢。在这个例子当中，所有的请求都需要带一个签名，签名值是由请求中带的参数根据规则使用 md5 方法计算出的值。我找不到一个 hook 可以让我在请求出去之前修改这个请求（添加上签名）。所以在这个例子当中，我的做法是根据 ngResource 的请求最后会使用 $httpBackend 这个底层服务，在 module 定义时我自己复制官方的相关代码，重新定义 $httpBackend 服务，在需要的地方做我自己的修改：

```
script.src = sign_url(url);
```

不错，我就改了这一句，但我不得不复制了 50 行官方源码到我的例子中。

另外一个需要说的是对返回数据的处理。因为 ngResource 会使用返回的数据直接填充实例，所以这个数据格式就很重要。

首先，我们可以使用 $http.defaults.transformResponse 来统一处理一下返回的数据，但是这并不能解决所有问题，可目前 ngResource 并不提供对每一个 action 的单独的后处理回调函数项。除非你的服务端是经过专门的适应性设计的，否则你用 ngResource 不可能爽。例子中，我为了获取当前列表的结果，我不得不自己去封装结果：

```
var list_list = List.getList(function(){
```

```
    var res = list_list[1];
    while(list_list.length > 0){list_list.pop()};
    angular.forEach(res.list, function(v){
      list_list.push(new List({list: v}));
    });
    $scope.list_list = list_list;
    $scope.show_add = true;
    return;
  });
```

16. AngularJS与其它框架的混用(jQuery, Dojo)

这个问题似乎很多人都关心，但是事实是，如果了解了 ng 的工作方式，这本来就不是一个问题了。

在我自己使用 ng 的过程当中，一直是混用 jQuery 的，以前还要加上一个 Dojo 。只要了解每种框架的工作方式，在具体的代码中每个框架都做了什么事，那么整体上控制起来就不会有问题。

回到 ng 上来看，首先对于 jQuery 来说，最开始说提到过，在 DOM 操作部分， ng 与 jQuery 是兼容的，如果没有 jQuery ， ng 自己也实现了兼容的部分 API 。

同时，最开始也提到过， ng 的使用最忌讳的一点就是修改 DOM 结构——你应该使用 ng 的模板机制进行数据绑定，以此来控制 DOM 结构，而不是直接操作。换句话来说，在不动 DOM 结构的这个前提之下，你的数据随便怎么改，随便使用哪个框架来控制都是没问题的，到时如有必要使用 $scope.$digest() 来通知 ng 一下即可。

下面这个例子，我们使用了 jQuery 中的 Deferred ($.ajax 就是返回一个 Deferred)，还使用了 ng 的 $timeout ，当然是在 ng 的结构之下：

```
<!DOCTYPE html>
<html ng-app="Demo">
<head>
<meta charset="utf-8" />
<title>AngularJS</title>
</head>
<body>

<div ng-controller="TestCtrl">
  <span ng-click="go()">{{ a }}</span>
</div>

<script type="text/javascript"
  src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js">
</script>
<script type="text/javascript"
  src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.3
/angular.min.js">
</script>

<script type="text/javascript">
var app = angular.module('Demo', [], angular.noop);
app.controller('TestCtrl', function($scope, $timeout){
  $scope.a = '点击我开始';

  var defer = $.Deferred();
  var f = function(){
    if($scope.a == ''){$scope.a = '已停止'; return}
    defer.done(function(){
      $scope.a.length < 10 ? $scope.a += '>' : $scope.a = '>';
      $timeout(f, 100);
    });
  }
  defer.done(function(){$scope.a = '>'; f()});

  $scope.go = function(){
    defer.resolve();
    $timeout(function(){$scope.a = ''}, 5000);
  }
});
</script>
</body>
</html>
```

再把 Dojo 加进来看与 DOM 结构相关的例子。之前说过，使用 ng 就最好不要手动修改 DOM 结构，但这里说两点：

1. 对于整个页面，你可以只在局部使用 ng ，不使用 ng 的地方你可以随意控制 DOM 。

2. 如果 DOM 结构有变动，你可以在 DOM 结构定下来之后再初始化 ng 。

下面这个例子使用了 AngularJS ， jQuery ， Dojo ：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>AngularJS</title>
<link rel="stylesheet"
  href="http://ajax.googleapis.com/ajax/libs/dojo/1.9.1/dijit/themes
/claro/claro.css" media="screen" />
</head>
<body class="claro">

<div ng-controller="TestCtrl" id="test_ctrl">

  <p ng-show="!btn_disable">
    <button ng-click="change()">调用dojo修改按钮</button>
  </p>

  <p id="btn_wrapper">
    <button data-dojo-type="dijit/form/Button" type="button">{{ a
}}</button>
  </p>

  <p>
    <input ng-model="dialog_text" ng-init="dialog_text='对话框内容'" />
    <button ng-click="dialog(dialog_text)">显示对话框</button>
  </p>

  <p ng-show="show_edit_text" style="display: none;">
    <span>需要编辑的内容:</span>
    <input ng-model="text" />
  </p>

  <div id="editor_wrapper">
    <div data-dojo-type="dijit/Editor" id="editor"></div>
  </div>

</div>


<script type="text/javascript"
  src="http://ajax.googleapis.com/ajax/libs/dojo/1.9.1/dojo/dojo.js">
</script>
<script type="text/javascript"
  src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js">
</script>
```

```
    <script type="text/javascript"
      src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.3
/angular.min.js">
    </script>

    <script type="text/javascript">

    require(['dojo/parser', 'dijit/Editor'], function(parser){
      parser.parse($('#editor_wrapper')[0]).then(function(){
        var app = angular.module('Demo', [], angular.noop);

        app.controller('TestCtrl', function($scope, $timeout){
          $scope.a = '我是ng, 也是dojo';
          $scope.show_edit_text = true;

          $scope.change = function(){
            $scope.a = 'DOM结构已经改变(不建议这样做)';
            require(['dojo/parser', 'dijit/form/Button', 'dojo/domReady!'],
              function(parser){
                parser.parse($('#btn_wrapper')[0]);
                $scope.btn_disable = true;
              }
            );
          }

          $scope.dialog = function(text){
            require(["dijit/Dialog", "dojo/domReady!"], function(Dialog){
              var dialog = new Dialog({
                  title: "对话框哦",
                  content: text,
                  style: "width: 300px"
              });
              dialog.show();
            });
          }

          require(['dijit/registry'], function(registry){
            var editor = registry.byId('editor');
            $scope.$watch('text', function(new_v){
              editor.setValue(new_v);
            });
          });

        });

        angular.bootstrap(document, ['Demo']);
      });

    });

    </script>
    </body>
    </html>
```

17. 自定义过滤器

先来回顾一下 ng 中的一些概念：

- module ，代码的组织单元，其它东西都是在定义在具体的模块中的。
- app ，业务概念，可能会用到多个模块。
- service ，仅在数据层面实现特定业务功能的代码封装。
- controller ，与 DOM 结构相关联的东西，即是一种业务封装概念，又体现了项目组织的层级结构。
- filter ，改变输入数据的一种机制。
- directive ，与 DOM 结构相关联的，特定功能的封装形式。

上面的这几个概念基本上就是 ng 的全部。每一部分都可以自由定义，使用时通过各要素的相互配合来实现我们的业务需求。

我们从最开始一致打交道的东西基本上都是 controller 层面的东西。在前面，也介绍了 module 和 service 的自定义。剩下的会介绍 filter 和 directive 的定义。基本上这几部分的定义形式都是一样的，原理上是通过 provider 来做注入形式的声明，在实际操作过程中，又有很多 shortcut 式的声明方式。

过滤器的自定义是最简单的，就是一个函数，接受输入，然后返回结果。在考虑过滤器时，我觉得很重要的一点： 无状态 。

具体来说，过滤器就是一个函数，函数的本质含义就是确定的输入一定得到确定的输出。虽然 filter 是定义在 module 当中的，而且 filter 又是在 controller 的 DOM 范围内使用的，但是，它和具体的 module ， controller ， scope 这些概念都没有关系（虽然在这里你可以使用 js 的闭包机制玩些花样），它仅仅是一个函数，而已。换句话说，它没有任何上下文关联的能力。

过滤器基本的定义方式：

```
var app = angular.module('Demo', [], angular.noop);
app.filter('map', function(){
  var filter = function(input){
    return input + '...';
  };
  return filter;
});
```

上面的代码定义了一个叫做 map 的过滤器。使用时：

```
<p>示例数据: {{ a|map }}</p>
```

过滤器也可以带参数，多个参数之间使用 : 分割，看一个完整的例子：

```
1  <div ng-controller="TestCtrl">
2  <p>示例数据: {{ a|map:map_value:'>>':'(no)' }}</p>
3  <p>示例数据: {{ b|map:map_value:'>>':'(no)' }}</p>
4  </div>
5
6
7  <script type="text/javascript">
8
9  var app = angular.module('Demo', [], angular.noop);
10  app.controller('TestCtrl', function($scope){
11    $scope.map_value = {
12      a: '一',
13      b: '二',
14      c: '三'
```

```
15    }
16    $scope.a = 'a';
17  });
18
19  app.filter('map', function(){
20    var filter = function(input, map_value, append, default_value){
21      var r = map_value[input];
22      if(r === undefined){ return default_value + append }
23      else { return r + append }
24    };
25    return filter;
26  });
27
28  angular.bootstrap(document, ['Demo']);
29  </script>
```

18. 自定义指令directive

这是 ng 最强大的一部分，也是最复杂最让人头疼的部分。

目前我们看到的所谓“模板”系统，只不过是官方实现的几个指令而已。这意味着，通过自定义各种指令，我们不但可以完全定义一套“模板”系统，更可以把 HTML 页面直接打造成为一种 DSL （领域特定语言）。

18.1. 指令的使用

使用指令时，它的名字可以有多种形式，把指令放在什么地方也有多种选择。

通常，指令的定义名是形如 ngBind 这样的 “camel cased” 形式。在使用时，它的引用名可以是：

- ng:bind
- ng_bind
- ng-bind
- x-ng-bind
- data-ng-bind

你可以根据你自己是否有 “HTML validator” 洁癖来选择。

指令可以放在多个地方，它们的作用相同：

- 作为标签的属性
- 作为标签类属性的值
- <my-dir></my-dir> 作为标签
- <!-- directive: my-dir exp --> 作为注释

这些方式可以使用指令定义中的 restrict 属性来控制。

可以看出，指令即可以作为标签使用，也可以作为属性使用。仔细考虑一下，这在类 XML 的结构当中真算得上是一种神奇的机制。

18.2. 指令的执行过程

ng 中对指令的解析与执行过程是这样的：

- 浏览器得到 HTML 字符串内容，解析得到 DOM 结构。
- ng 引入，把 DOM 结构扔给 $compile 函数处理：
  - 找出 DOM 结构中有变量占位符
  - 匹配找出 DOM 中包含的所有指令引用
  - 把指令关联到 DOM
  - 关联到 DOM 的多个指令按权重排列
  - 执行指令中的 compile 函数（改变 DOM 结构，返回 link 函数）

- 得到的所有 link 函数组成一个列表作为 $compile 函数的返回
- 执行 link 函数（连接模板的 scope）。

18.3. 基本的自定义方法

自定义一个指令可以非常非常的复杂，但是其基本的调用形式，同自定义服务大概是相同的：

```
<p show style="font-size: 12px;"></p>

<script type="text/javascript">

var app = angular.module('Demo', [], angular.noop);

app.directive('show', function(){
  var func = function($scope, $element, $attrs){
    console.log($scope);
    console.log($element);
    console.log($attrs);
  }
  return func;
  //return {compile: function(){return func}}
});

angular.bootstrap(document, ['Demo']);
</script>
```

如果在 directive 中直接返回一个函数，则这个函数会作为 compile 的返回值，也即是作为 link 函数使用。这里说的 compile 和 link 都是一个指令

的组成部分，一个完整的定义应该返回一个对象，这个对象包括了多个属性：

- name
- *priority*
- *terminal*
- *scope*
- *controller*
- *require*
- *restrict*
- *template*
- *templateUrl*
- *replace*
- *transclude*
- *compile*
- *link*

上面的每一个属性，都可以单独探讨的。

下面是一个完整的基本的指令定义例子：

```
<code lines>
//失去焦点使用 jQuery 的扩展支持冒泡
app.directive('ngBlur', function($parse){
  return function($scope, $element, $attr){
    var fn = $parse($attr['ngBlur']);
    $element.on('focusout', function(event){
      fn($scope, {$event: event});
```

```
    });
  }
});
</code>

<div code lines>
//失去焦点使用 jQuery 的扩展支持冒泡
app.directive('ngBlur', function($parse){
  return function($scope, $element, $attr){
    var fn = $parse($attr['ngBlur']);
    $element.on('focusout', function(event){
      fn($scope, {$event: event});
    });
  }
});
</div>
1  var app = angular.module('Demo', [], angular.noop);
2
3  app.directive('code', function(){
4    var func = function($scope, $element, $attrs){
5
6      var html = $element.text();
7      var lines = html.split('\n');
8
```

```
 9      //处理首尾空白
10      if(lines[0] == ''){lines = lines.slice(1, lines.length - 1)}
11      if(lines[lines.length-1] == ''){lines = lines.slice(0, lines.length - 1)}
12
13      $element.empty();
14
15      //处理外框
16      (function(){
17        $element.css('clear', 'both');
18        $element.css('display', 'block');
19        $element.css('line-height', '20px');
20        $element.css('height', '200px');
21      })();
22
23      //是否显示行号的选项
24      if('lines' in $attrs){
25        //处理行号
26        (function(){
27          var div = $('<div style="width: %spx; background-color: gray;
float: left; text-align: right; padding-right: 5px; margin-right: 10px;"></div>'
28                      .replace('%s', String(lines.length).length * 10));
29          var s = '';
30          angular.forEach(lines, function(_, i){
31            s += '<pre style="margin: 0;">%s</pre>\n'.replace('%s', i +
1);
```

```
32          });
33          div.html(s);
34          $element.append(div);
35        })();
36      }
37
38      //处理内容
39      (function(){
40        var div = $('<div style="float: left;"></div>');
41        var s = '';
42        angular.forEach(lines, function(l){
43          s += '<span style="margin: 0;">%s</span><br
/>\n'.replace('%s', l.replace(/\s/g, '<span> </span>'));
44        });
45        div.html(s);
46        $element.append(div);
47      })();
48    }
49
50    return {link: func,
51            restrict: 'AE'}; //以元素或属性的形式使用命令
52  });
53
54  angular.bootstrap(document, ['Demo']);
```

上面这个自定义的指令，做的事情就是解析节点中的文本内容，然后修改它，再把生成的新内容填充到节点当中去。其间还涉及了节点属性值 lines 的处理。这算是指令中最简单的一种形式。因为它是“一次性使用”，中间没有变量的处理。比如如果节点原来的文本内容是一个变量引用，类似于 {{ code }} ，那上面的代码就不行了。这种情况麻烦得多。后面会讨论。

18.4. 属性值类型的自定义

官方代码中的 ng-show 等算是我说的这种类型。使用时主要是在节点加添加一个属性值以附加额外的功能。看一个简单的例子：

```
<p color="red">有颜色的文本</p>

<color color="red">有颜色的文本</color>


<script type="text/javascript">


var app = angular.module('Demo', [], angular.noop);

app.directive('color', function(){
  var link = function($scope, $element, $attrs){
    $element.css('color', $attrs.color);
  }
  return {link: link,
          restrict: 'AE'};
});

angular.bootstrap(document, ['Demo']);
</script>
```

我们定义了一个叫 color 的指令，可以指定节点文本的颜色。但是这个例子还无法像 ng-show 那样工作的，这个例子只能渲染一次，然后就无法根据变量来重新改变显示了。要响应变化，我们需要手工使用 scope 的 $watch 来处理：

```
1
2  <div ng-controller="TestCtrl">
3    <p color="color">有颜色的文本</p>
4    <p color="'blue'">有颜色的文本</p>
5  </div>
6
7  <script type="text/javascript">
8
9  var app = angular.module('Demo', [], angular.noop);
10
11 app.directive('color', function(){
12   var link = function($scope, $element, $attrs){
13     $scope.$watch($attrs.color, function(new_v){
14       $element.css('color', new_v);
15     });
16   }
17   return link;
18 });
19
20 app.controller('TestCtrl', function($scope){
21   $scope.color = 'red';
```

```
22  });
23
24  angular.bootstrap(document, ['Demo']);
25  </script>
```

18.5. Compile的细节

指令的处理过程，是 ng 的 Compile 过程的一部分，它们也是紧密联系的。继续深入指令的定义方法，首先就要对 Compile 的过程做更细致的了解。

前面说过， ng 对页面的处理过程：

- 浏览器把 HTML 字符串解析成 DOM 结构。
- ng 把 DOM 结构给 $compile ， 返回一个 link 函数。
- 传入具体的 scope 调用这个 link 函数。
- 得到处理后的 DOM ， 这个 DOM 处理了指令，连接了数据。

$compile 最基本的使用方式：

```
var link = $compile('<p>{{ text }}</p>');
var node = link($scope);
console.log(node);
```

上面的 $compile 和 link 调用时都有额外参数来实现其它功能。先看 link 函数，它形如：

```
function(scope[, cloneAttachFn]
```

第二个参数 cloneAttachFn 的作用是，表明是否复制原始节点，及对复制节点需要做的处理，下面这个例子说明了它的作用：

```
<div ng-controller="TestCtrl"></div>
<div id="a">A {{ text }}</div>
<div id="b">B </div>
```

```
app.controller('TestCtrl', function($scope, $compile){
  var link = $compile($('#a'));

  //true参数表示新建一个完全隔离的scope,而不是继承的child scope
  var scope = $scope.$new(true);
  scope.text = '12345';

  //var node = link(scope, function(){});
  var node = link(scope);

  $('#b').append(node);
});
```

cloneAttachFn 对节点的处理是有限制的，你可以添加 class ， 但是不能做与数据绑定有关的其它修改（修改了也无效）：

```
app.controller('TestCtrl', function($scope, $compile){
  var link = $compile($('#a'));
  var scope = $scope.$new(true);
  scope.text = '12345';

  var node = link(scope, function(clone_element, scope){
    clone_element.text(clone_element.text() + ' ...'); //无效
    clone_element.text('{{ text2 }}'); //无效
    clone_element.addClass('new_class');
  });
```

```
    $('#b').append(node);
  });
```

修改无效的原因是，像 {{ text }} 这种所谓的 Interpolate 在 $compile 中已经被处理过了，生成了相关函数（这里起作用的是 directive 中的一个 postLink 函数），后面执行 link 就是执行了 $compile 生成的这些函数。当然，如果你的文本没有数据变量的引用，那修改是会有效果的。

前面在说自定义指令时说过， link 函数是由 compile 函数返回的，也就像前面说的，应该把改变 DOM 结构的逻辑放在 compile 函数中做。

$compile 还有两个额外的参数：

```
$compile(element, transclude, maxPriority);
```

maxPriority 是指令的权重限制，这个容易理解，后面再说。

transclude 是一个函数，这个函数会传递给 compile 期间找到的 directive 的 compile 函数（编译节点的过程中找到了指令，指令的 compile 函数会接受编译时传递的 transclude 函数作为其参数）。

但是在实际使用中，除我们手工在调用 $compile 之外，初始化时的根节点 compile 是不会传递这个参数的。

在我们定义指令时，它的 compile 函数是这个样子的：

```
function compile(tElement, tAttrs, transclude) { ... }
```

事实上， transclude 的值，就是 directive 所在的 原始 节点，把原始节点重新做了编译之后得到的 link 函数（需要 directive 定义时使用 transclude 选项），后面会专门演示这个过程。所以，官方文档上也把 transclude 函数描述成 link 函数的样子（如果自定义的指令只用在自己手动 $compile 的环境中，那这个函数的形式是可以随意的）：

```
{function(angular.Scope[, cloneAttachFn]}
```

所以记住，定义指令时， compile 函数的第三个参数 transclude ，就是一个 link ，装入 scope 执行它你就得到了一个节点。

18.6. transclude的细节

transclude 有两方面的东西，一个是使用 $compile 时传入的函数，另一个是定义指令的 compile 函数时接受的一个参数。虽然这里的一出一进本来是相互对应的，但是实际使用中，因为大部分时候不会手动调用 $compile，所以，在“默认”情况下，指令接受的 transclude 又会是一个比较特殊的函数。

看一个基本的例子：

```
var app = angular.module('Demo', [], angular.noop);


app.directive('more', function(){
  var func = function(element, attrs, transclude){
    var sum = transclude(1, 2);
    console.log(sum);
    console.log(element);
  }

  return {compile: func,
          restrict: 'E'};
});

app.controller('TestCtrl', function($scope, $compile, $element){
  var s = '<more>123</more>';
  var link = $compile(s, function(a, b){return a + b});
  var node = link($scope);
  $element.append(node);
```

```
});


angular.bootstrap(document, ['Demo']);
```

我们定义了一个 more 指令，它的 compile 函数的第三个参数，就是我们手工 $compile 时传入的。

如果不是手工 $compile ，而是 ng 初始化时找出的指令，则 transclude 是一个 link 函数（指令定义需要设置 transclude 选项）：

```
<div more>123</div>
app.directive('more', function($rootScope, $document){
  var func = function(element, attrs, link){
    var node = link($rootScope);
    node.removeAttr('more'); //不去掉就变死循环了
    $('body', $document).append(node);
  }

  return {compile: func,
        transclude: 'element', // element是节点没,其它值是节点的内容没
        restrict: 'A'};
});
```

18.7. 把节点内容作为变量处理的类型

回顾最开始的那个代码显示的例子，那个例子只能处理一次节点内容。如果节点的内容是一个变量的话，需要用另外的思路来考虑。这里我们假设的例子是，定义一个指令 showLenght ，它的作用是在一段文本的开头显示出这段节点文本的长度，节点文本是一个变量。指令使用的形式是：

```
<div ng-controller="TestCtrl">
```

```
    <div show-length>{{ text }}</div>
    <button ng-click="text='xx'">改变</button>
  </div>
```

从上面的 HTML 代码中，大概清楚 ng 解析它的过程（只看 show-length 那一行）：

- 解析 div 时发现了一个 show-length 的指令。
- 如果 show-length 指令设置了 transclude 属性，则 div 的节点内容被重新编译，得到的 link 函数作为指令 compile 函数的参数传入。
- 如果 show-length 指令没有设置 transclude 属性，则继续处理它的子节点（ TextNode ）。
- 不管是上面的哪种情况，都会继续处理到 {{ text }} 这段文本。
- 发现 {{ text }} 是一个 Interpolate ，于是自动在此节点中添加了一个指令，这个指令的 link 函数就是为 scope 添加了一个 $watch ，实现的功能是是当 scope 作 $digest 的时候，就更新节点文本。

与处理 {{ text }} 时添加的指令相同，我们实现 showLength 的思路，也就是：

- 修改原来的 DOM 结构
- 为 scope 添加 $watch ，当 $digest 时修改指定节点的文本，其值为指定节点文本的长度。

代码如下：

```
app.directive('showLength', function($rootScope, $document){
  var func = function(element, attrs, link){


    return function(scope, ielement, iattrs, controller){
      var node = link(scope);
      ielement.append(node);
```

```
    var lnode = $('<span></span>');
    ielement.prepend(lnode);

    scope.$watch(function(scope){
      lnode.text(node.text().length);
    });
  };
}

return {compile: func,
        transclude: true, // element是节点没,其它值是节点的内容没
        restrict: 'A'};
});
```

上面代码中，因为设置了 transclude 属性，我们在 showLength 的 link 函数（就是 return 的那个函数）中，使用 func 的第三个函数来重塑了原来的文本节点，并放在我们需要的位置上。然后，我们添加自己的节点来显示长度值。最后给当前的 scope 添加 $watch ，以更新这个长度值。

18.8. 指令定义时的参数

指令定义时的参数如下:

- name
- priority
- terminal
- scope
- controller
- require

- restrict
- template
- templateUrl
- replace
- transclude
- compile
- link

现在我们开始一个一个地吃掉它们……，但是并不是按顺序讲的。

priority

这个值设置指令的权重，默认是 0 。当一个节点中有多个指令存在时，就按着权限从大到小的顺序依次执行它们的 compile 函数。相同权重顺序不定。

terminal

是否以当前指令的权重为结束界限。如果这值设置为 true ， 则节点中权重小于当前指令的其它指令不会被执行。相同权重的会执行。

restrict

指令可以以哪些方式被使用，可以同时定义多种方式。

- E 元素方式 <my-directive></my-directive>
- A 属性方式
- C 类方式
- M 注释方式 <!-- directive: my-directive exp -->

transclude

前面已经讲过基本的用法了。可以是 'element' 或 true 两种值。

compile

基本的定义函数。 function compile(tElement, tAttrs, transclude) { ... }

link

前面介绍过了。大多数时候我们不需要单独定义它。只有 compile 未定义时 link 才会被尝试。 function link(scope, iElement, iAttrs, controller) { ... }

scope

scope 的形式。 false 节点的 scope， true 继承创建一个新的 scope，{} 不继承创建一个新的隔离 scope 。 {@attr: '引用节点属性', =attr: '把节点属性值引用成scope属性值', &attr: '把节点属性值包装成函数'}

controller

为指令定义一个 controller， function controller($scope, $element, $attrs, $transclude) { ... }

name

指令的 controller 的名字，方便其它指令引用。

require

要引用的其它指令 conroller 的名字， ?name 忽略不存在的错误，^name 在父级查找。

template

模板内容。

templateUrl

从指定地址获取模板内容。

replace

是否使用模板内容替换掉整个节点， true 替换整个节点， false 替换节点内容。

<a b>

var app = angular.module('Demo', [], angular.noop);

```
app.directive('a', function(){
  var func = function(element, attrs, link){
    console.log('a');
  }

  return {compile: func,
          priority: 1,
          restrict: 'EA'};
});

app.directive('b', function(){
  var func = function(element, attrs, link){
    console.log('b');
  }

  return {compile: func,
          priority: 2,
          //terminal: true,
          restrict: 'A'};
 });
```

上面几个参数值都是比较简单且容易理想的。

再看 scope 这个参数：

```
 <div ng-controller="TestCtrl">
   <div a b></div>
```

```
 </div>
var app = angular.module('Demo', [], angular.noop);

app.directive('a', function(){
  var func = function(element, attrs, link){
    return function(scope){
      console.log(scope);
    }
  }

  return {compile: func,
          scope: true,
          restrict: 'A'};
});

app.directive('b', function(){
  var func = function(element, attrs, link){
    return function(scope){
      console.log(scope);
    }
  }

  return {compile: func,
          restrict: 'A'};
```

```
24  });
25
26  app.controller('TestCtrl', function($scope){
27    $scope.a = '123';
28    console.log($scope);
29  });
```

对于 scope：

- 默认为 false， link 函数接受的 scope 为节点所在的 scope 。
- 为 true 时，则 link 函数中第一个参数（还有 controller 参数中的 $scope ）， scope 是节点所在的 scope 的 child scope ，并且如果节点中有多个指令，则只要其中一个指令是 true 的设置，其它所有指令都会受影响。

这个参数还有其它取值。当其为 {} 时，则 link 接受一个完全隔离（isolate）的 scope ，于 true 的区别就是不会继承其它 scope 的属性。但是这时，这个 scope 的属性却可以有很灵活的定义方式：

@attr 引用节点的属性。

```
<div ng-controller="TestCtrl">
  <div a abc="here" xx="{{ a }}" c="ccc"></div>
</div>
var app = angular.module('Demo', [], angular.noop);


app.directive('a', function(){
  var func = function(element, attrs, link){
    return function(scope){
      console.log(scope);
```

```
    }
  }

  return {compile: func,
          scope: {a: '@abc', b: '@xx', c: '@'},
          restrict: 'A'};
});

app.controller('TestCtrl', function($scope){
  $scope.a = '123';
});
```

- @abc 引用 div 节点的 abc 属性。
- @xx 引用 div 节点的 xx 属性，而 xx 属性又是一个变量绑定，于是 scope 中 b 属性值就和 TestCtrl 的 a 变量绑定在一起了。
- @ 没有写 attr name ，则默认取自己的值，这里是取 div 的 c 属性。

=attr 相似，只是它把节点的属性值当成节点 scope 的属性名来使用，作用相当于上面例子中的 @xx：

```
<div ng-controller="TestCtrl">
  <div a abc="here"></div>
</div>
var app = angular.module('Demo', [], angular.noop);

app.directive('a', function(){
  var func = function(element, attrs, link){
```

```
    return function(scope){
      console.log(scope);
    }
  }

  return {compile: func,
          scope: {a: '=abc'},
          restrict: 'A'};
});

app.controller('TestCtrl', function($scope){
  $scope.here = '123';
});
```

&attr 是包装一个函数出来，这个函数以节点所在的 scope 为上下文。来看一个很爽的例子：

```
<div ng-controller="TestCtrl">
  <div a abc="here = here + 1" ng-click="show(here)">这里</div>
  <div>{{ here }}</div>
</div>
```

```
1  var app = angular.module('Demo', [], angular.noop);
2
3  app.directive('a', function(){
4    var func = function(element, attrs, link){
5      return function llink(scope){
```

```
6         console.log(scope);
7         scope.a();
8         scope.b();
9
10         scope.show = function(here){
11           console.log('Inner, ' + here);
12           scope.a({here: 5});
13         }
14       }
15     }
16
17     return {compile: func,
18             scope: {a: '&abc', b: '&ngClick'},
19             restrict: 'A'};
20   });
21
22   app.controller('TestCtrl', function($scope){
23     $scope.here = 123;
24     console.log($scope);
25
26     $scope.show = function(here){
27       console.log(here);
28     }
29   });
```

scope.a 是 &abc ， 即：

```
scope.a = function(){here = here + 1}
```

只是其中的 here 是 TestCtrl 的。

scope.b 是 &ngClick ， 即：

```
scope.b = function(){show(here)}
```

这里的 show() 和 here 都是 TestCtrl 的，于是上面的代码最开始会在终端输出一个 124 。

当点击“这里”时，这时执行的 show(here) 就是 llink 中定义的那个函数了，与 TestCtrl 无关。但是，其间的 scope.a({here:5}) ，因为 a 执行时是 TestCtrl 的上下文，于是向 a 传递的一个对象，里面的所有属性 TestCtrl 就全收下了，接着执行 here=here+1 ，于是我们会在屏幕上看到 6 。

这里是一个上下文交错的环境，通过 & 这种机制，让指令的 scope 与节点的 scope 发生了互动。真是鬼斧神工的设计。而实现它，只用了几行代码：

```
case '&': {
  parentGet = $parse(attrs[attrName]);
  scope[scopeName] = function(locals) {
    return parentGet(parentScope, locals);
  }
  break;
}
```

再看 controller 这个参数。这个参数的作用是提供一个 controller 的构造函数，它会在 compile 函数之后， link 函数之前被执行。

```
<a>haha</a>
1  var app = angular.module('Demo', [], angular.noop);
2
```

```
app.directive('a', function(){
  var func = function(){
    console.log('compile');
    return function(){
      console.log('link');
    }
  }

  var controller = function($scope, $element, $attrs, $transclude){
    console.log('controller');
    console.log($scope);

    var node = $transclude(function(clone_element, scope){
      console.log(clone_element);
      console.log('--');
      console.log(scope);
    });
    console.log(node);
  }

  return {compile: func,
          controller: controller,
          transclude: true,
          restrict: 'E'}
```

```
27  });
```

controller 的最后一个参数， $transclude ， 是一个只接受 cloneAttachFn 作为参数的一个函数。

按官方的说法，这个机制的设计目的是为了让各个指令之间可以互相通信。参考普通节点的处理方式，这里也是处理指令 scope 的合适位置。

```
 <a b>kk</a>
1  var app = angular.module('Demo', [], angular.noop);
2
3  app.directive('a', function(){
4    var func = function(){
5    }
6
7    var controller = function($scope, $element, $attrs, $transclude){
8      console.log('a');
9      this.a = 'xx';
10   }
11
12   return {compile: func,
13           name: 'not_a',
14           controller: controller,
15           restrict: 'E'}
16 });
17
18 app.directive('b', function(){
19   var func = function(){
```

```
20      return function($scope, $element, $attrs, $controller){
21        console.log($controller);
22      }
23    }
24
25    var controller = function($scope, $element, $attrs, $transclude){
26      console.log('b');
27    }
28
29    return {compile: func,
30            controller: controller,
31            require: 'not_a',
32            restrict: 'EA'}
33    });
```

name 参数在这里可以用以为 controller 重起一个名字，以方便在 require 参数中引用。

require 参数可以带两种前缀（可以同时使用）：

- ？，如果指定的 controller 不存在，则忽略错误。即：

```
require: '?not_b'
```

-

如果名为 not_b 的 controller 不存在时，不会直接抛出错误， link 函数中对应的 $controller 为 undefined 。

- ^，同时在父级节点中寻找指定的 controller ，把上面的例子小改一下：

```
<a><b>kk</b></a>
```

-

把 a 的 require 改成（否则就找不到 not_a 这个 controller ）：
require: '?^not_a'

-

还剩下几个模板参数：

template 模板内容，这个内容会根据 replace 参数的设置替换节点或只替换节点内容。

templateUrl 模板内容，获取方式是异步请求。

replace 设置如何处理模板内容。为 true 时为替换掉指令节点，否则只替换到节点内容。

```
<div ng-controller="TestCtrl">
  <h1 a>原始内容</h1>
</div>
var app = angular.module('Demo', [], angular.noop);

app.directive('a', function(){
  var func = function(){
  }

  return {compile: func,
```

```
            template: '<p>标题 {{ name }} <button ng-
click="name=\'hahaha\'">修改</button></p>',
            //replace: true,
            //controller: function($scope){$scope.name = 'xxx'},
            //scope: {},
            scope: true ,
            controller: function($scope){console.log($scope)},
            restrict: 'A'}
    });

    app.controller('TestCtrl', function($scope){
      $scope.name = '123';
      console.log($scope);
    });
```

template 中可以包括变量引用的表达式，其 scope 遵寻 scope 参数的作用（可能受继承关系影响）。

templateUrl 是异步请求模板内容，并且是获取到内容之后才开始执行指令的 compile 函数。

最后说一个 compile 这个参数。它除了可以返回一个函数用为 link 函数之外，还可以返回一个对象，这个对象能包括两个成员，一个 pre ，一个 post 。实际上， link 函数是由两部分组成，所谓的 preLink 和 postLink 。区别在于执行顺序，特别是在指令层级嵌套的结构之下， postLink 是在所有的子级指令 link 完成之后才最后执行的。 compile 如果只返回一个函数，则这个函数被作为 postLink 使用：

```
<a><b></b></a>
```

```
1   var app = angular.module('Demo', [], angular.noop);
```

```
2
3   app.directive('a', function(){
4     var func = function(){
5       console.log('a compile');
6       return {
7         pre: function(){console.log('a link pre')},
8         post: function(){console.log('a link post')},
9       }
10    }
11
12    return {compile: func,
13            restrict: 'E'}
14  });
15
16  app.directive('b', function(){
17    var func = function(){
18      console.log('b compile');
19      return {
20        pre: function(){console.log('b link pre')},
21        post: function(){console.log('b link post')},
22      }
23    }
24
25    return {compile: func,
```

```
26            restrict: 'E'}
27   });
```

18.9. Attributes的细节

节点属性被包装之后会传给 compile 和 link 函数。从这个操作中，我们可以得到节点的引用，可以操作节点属性，也可以为节点属性注册侦听事件。

```
<test a="1" b c="xxx"></test>
var app = angular.module('Demo', [], angular.noop);


app.directive('test', function(){
  var func = function($element, $attrs){
    console.log($attrs);
  }
  return {compile: func,
          restrict: 'E'}
```

整个 Attributes 对象是比较简单的，它的成员包括了：

$$element 属性所在的节点。

$attr 所有的属性值（类型是对象）。

$normalize 一个名字标准化的工具函数，可以把 ng-click 变成 ngClick。

$observe 为属性注册侦听器的函数。

$set 设置对象属性，及节点属性的工具。

除了上面这些成员，对象的成员还包括所有属性的名字。

先看 $observe 的使用，基本上相当于 $scope 中的 $watch：

```
<div ng-controller="TestCtrl">
  <test a="{{ a }}" b c="xxx"></test>
  <button ng-click="a=a+1">修改</button>
</div>
var app = angular.module('Demo', [], angular.noop);

app.directive('test', function(){
  var func = function($element, $attrs){
    console.log($attrs);

    $attrs.$observe('a', function(new_v){
      console.log(new_v);
    });
  }

  return {compile: func,
          restrict: 'E'}
});
```

```
app.controller('TestCtrl', function($scope){
  $scope.a = 123;
});
```

$set 方法的定义是： function(key, value, writeAttr, attrName) { ... } 。

- key 对象的成员名。
- value 需要设置的值。
- writeAttr 是否同时修改 DOM 节点的属性（注意区别“节点”与“对象”），默认为 true 。
- attrName 实际的属性名，与“标准化”之后的属性名有区别。

```
<div ng-controller="TestCtrl">
  <test a="1" ys-a="123" ng-click="show(1)">这里</test>
</div>
var app = angular.module('Demo', [], angular.noop);

app.directive('test', function(){
  var func = function($element, $attrs){
    $attrs.$set('b', 'ooo');
    $attrs.$set('a-b', '11');
    $attrs.$set('c-d', '11', true, 'c_d');
    console.log($attrs);
  }

  return {compile: func,
          restrict: 'E'}
```

```
});


app.controller('TestCtrl', function($scope){
  $scope.show = function(v){console.log(v);}
});
```

从例子中可以看到，原始的节点属性值对，放到对象中之后，名字一定是“标准化”之后的。但是手动 $set 的新属性，不会自动做标准化处理。

18.10. 预定义的 NgModelController

在前面讲 conroller 参数的时候，提到过可以为指令定义一个 conroller 。官方的实现中，有很多已定义的指令，这些指令当中，有两个已定义的 controller ，它们是 NgModelController 和 FormController ，对应 ng-model 和 form 这两个指令（可以参照前面的“表单控件”一章）。

在使用中，除了可以通过 $scope 来取得它们的引用之外，也可以在自定义指令中通过 require 参数直接引用，这样就可以在 link 函数中使用 controller 去实现一些功能。

先看 NgModelController 。这东西的作用有两个，一是控制 ViewValue 与 ModelValue 之间的转换关系（你可以实现看到的是一个值，但是存到变量里变成了另外一个值），二是与 FormController 配合做数据校验的相关逻辑。

先看两个应该是最有用的属性：

$formatters 是一个由函数组成的列表，串行执行，作用是把变量值变成显示的值。

$parsers 与上面的方向相反，把显示的值变成变量值。

假设我们在变量中要保存一个列表的类型，但是显示的东西只能是字符串，所以这两者之间需要一个转换：

```
<div ng-controller="TestCtrl">
  <input type="text" ng-model="a" test />
  <button ng-click="show(a)">查看</button>
</div>
```

```
var app = angular.module('Demo', [], angular.noop);

app.directive('test', function(){
  var link = function($scope, $element, $attrs, $ctrl){

    $ctrl.$formatters.push(function(value){
      return value.join(',');
    });

    $ctrl.$parsers.push(function(value){
      return value.split(',');
    });
  }

  return {compile: function(){return link},
          require: 'ngModel',
          restrict: 'A'}
});

app.controller('TestCtrl', function($scope){
```

```
21    $scope.a = [];
22    //$scope.a = [1,2,3];
23    $scope.show = function(v){
24      console.log(v);
25    }
26  });
```

上面在定义 test 这个指令， require 参数指定了 ngModel 。同时因为 DOM 结构， ng-model 是存在的。于是， link 函数中就可以获取到一个 NgModelController 的实例，即代码中的 $ctrl 。

我们添加了需要的过滤函数：

- 从变量(ModelValue)到显示值(ViewValue)的过程， $formatters 属性，把一个列表变成一个字符串。
- 从显示值到变量的过程， $parsers 属性，把一个字符串变成一个列表。

对于显示值和变量，还有其它的 API ，这里就不细说了。

另一部分，是关于数据校验的，放到下一章同 FormController 一起讨论。

18.11. 预定义的 FormController

前面的“表单控制”那章，实际上讲的就是 FormController ，只是那里是从 scope 中获取到的引用。现在从指令定义的角度，来更清楚地了解 Form-Controller 及 NgModelController 是如何配合工作的。

先说一下， form 和 ngForm 是官方定义的两个指令，但是它们其实是同一个东西。前者只允许以标签形式使用，而后者允许 EAC 的形式。DOM 结构中， form 标签不能嵌套，但是 ng 的指令没有这个限制。不管是 form 还是 ngForm ，它们的 controller 都被命名成了 form 。 所以 require 这个参数不要写错了。

FormController 的几个成员是很好理解的：

$pristine 表单是否被动过

$dirty 表单是否没被动过

$valid 表单是否检验通过

$invalid 表单是否检验未通过

$error 表单中的错误

$setDirty() 直接设置 $dirty 及 $pristine

```
<div ng-controller="TestCtrl">
  <div ng-form test>
    <input ng-model="a" type="email" />
    <button ng-click="do()">查看</button>
  </div>
</div>
var app = angular.module('Demo', [], angular.noop);

app.directive('test', function(){
  var link = function($scope, $element, $attrs, $ctrl){
    $scope.do = function(){
      //$ctrl.$setDirty();
```

```
        console.log($ctrl.$pristine); //form是否没被动过
        console.log($ctrl.$dirty); //form是否被动过
        console.log($ctrl.$valid); //form是否被检验通过
        console.log($ctrl.$invalid); //form是否有错误
        console.log($ctrl.$error); //form中有错误的字段
      }
    }

    return {compile: function(){return link},
            require: 'form',
            restrict: 'A'}
});
```

```
app.controller('TestCtrl', function($scope){
});
```

$error 这个属性，是一个对象， key 是错误名， value 部分是一个列表，其成员是对应的 NgModelController 的实例。

FormController 可以自由增减它包含的那些，类似于 NgModelController 的实例。在 DOM 结构上，有 ng-model 的 input 节点的 NgMoelController 会被自动添加。

$addControl() 添加一个 conroller

$removeControl() 删除一个 controller

这两个手动使用机会应该不会很多。被添加的实例也可以手动实现所有的 NgModelController 的方法

```
<div ng-controller="TestCtrl">
  <bb />
  <div ng-form test>
    <input ng-model="a" type="email" />
    <button ng-click="add()">添加</button>
  </div>
</div>
```

```
1  var app = angular.module('Demo', [], angular.noop);
2
3  app.directive('test', function(){
4    var link = function($scope, $element, $attrs, $ctrl){
5      $scope.add = function(){
6        $ctrl.$addControl($scope.bb);
7        console.log($ctrl);
8      }
9    }
10
11   return {compile: function(){return link},
12           require: 'form',
13           restrict: 'A'}
14 });
15
```

```
16  app.directive('bb', function(){
17    var controller = function($scope, $element, $attrs, $transclude){
18      $scope.bb = this;
19      this.$name = 'bb';
20    }
21
22    return {compile: angular.noop,
23            restrict: 'E',
24            controller: controller}
25  });
26
27  app.controller('TestCtrl', function($scope){
28  });
```

整合 FormController 和 NgModelController 就很容易扩展各种类型的字段:

```
<div ng-controller="TestCtrl">
  <form name="f">
    <input type="my" ng-model="a" />
    <button ng-click="show()">查看</button>
  </form>
</div>
```

```
1  var app = angular.module('Demo', [], angular.noop);
2
3  app.directive('input', function(){
```

```
4     var link = function($scope, $element, $attrs, $ctrl){
5       console.log($attrs.type);
6       var validator = function(v){
7         if(v == '123'){
8           $ctrl.$setValidity('my', true);
9           return v;
10        } else {
11          $ctrl.$setValidity('my', false);
12          return undefined;
13        }
14      }
15
16      $ctrl.$formatters.push(validator);
17      $ctrl.$parsers.push(validator);
18    }
19
20    return {compile: function(){return link},
21            require: 'ngModel',
22            restrict: 'E'}
23  });
24
25  app.controller('TestCtrl', function($scope){
26      $scope.show = function(){
27        console.log($scope.f);
```

```
28      }
29  });
```

虽然官方原来定义了几种 type ，但这不妨碍我们继续扩展新的类型。如果新的 type 参数值不在官方的定义列表里，那会按 text 类型先做处理，这其实什么影响都没有。剩下的，就是写我们自己的验证逻辑就行了。

上面的代码是参见官方的做法，使用格式化的过程，同时在里面做有效性检查。

18.12. 示例：文本框

这个例子与官网上的那个例子相似。最终是要显示一个文本框，这个文本框由标题和内容两部分组成。而且标题和内容则是引用 controller 中的变量值。

HTML 部分的代码：

```
<div ng-controller="TestCtrl">
  <ys-block title="title" text="text"></ys-block>
  <p>标题: <input ng-model="title" /></p>
  <p>内容: <input ng-model="text" /></p>
  <ys-block title="title" text="text"></ys-block>
</div>
```

从这个期望实现效果的 HTML 代码中，我们可以考虑设计指令的实现方式：

- 这个指令的使用方式是“标签”， 即 restrict 这个参数应该设置为 E 。
- 节点的属性值是对 controller 变量的引用，那么我们应该在指令的 scope 中使用 = 的方式来指定成员值。
- 最终的效果显示需要进行 DOM 结构的重构，那直接使用 template 就好了。

- 自定义的标签在最终效果中是多余的，所有 replace 应该设置为 true 。

JS 部分的代码：

```
var app = angular.module('Demo', [], angular.noop);


app.directive('ysBlock', function(){
  return {compile: angular.noop,
        template: '<div style="width: 200px; border: 1px solid black;"><h1 style="background-color: gray; color: white; font-size: 22px;">{{ title }}</h1><div>{{ text }}</div></div>',
        replace: true,
        scope: {title: '=title', text: '=text'},
        restrict: 'E'};
});


app.controller('TestCtrl', function($scope){
  $scope.title = '标题在这里';
  $scope.text = '内容在这里';
});


angular.bootstrap(document, ['Demo']);
```

可以看到，这种简单的组件式指令，只需要作 DOM 结构的变换即可实现，连 compile 函数都不需要写。

18.13. 示例：模板控制语句 for

这个示例尝试实现一个重复语句，功能同官方的 ngRepeat ，但是使用方式类似于我们通常编程语言中的 for 语句：

```
<div ng-controller="TestCtrl" ng-init="obj_list=[1,2,3,4]; name='name'">
  <ul>
    <for o in obj_list>
      <li>{{ o }}, {{ name }}</li>
    </for>
  </ul>
  <button ng-click="obj_list=[1,2]; name='o?'">修改</button>
</div>
```

同样，我们从上面的使用方式去考虑这个指令的实现：

- 这是一个完全的控制指令，所以单个节点应该只有它一个指令起作用就好了，于是权重要比较高，并且“到此为止”—— priority 设置为 1000 ，terminal 设置为 true 。
- 使用时的语法问题。事实上浏览器会把 for 节点补充成一个正确的 HTML 结构，即里面的属性都会变成类似 o="" 这样。我们通过节点的 outerHTML 属性取到字符串并解析取得需要的信息。
- 我们把 for 节点之间的内容作为一个模板，并且通过循环多次渲染该模板之后把结果填充到合适的位置。
- 在处理上面的那个模板时，需要不断地创建新 scope 的，并且 o 这个成员需要单独赋值。

注意：这里只是简单实现功能。官方的那个 ngRepeat 比较复杂，是做了专门的算法优化的。当然，这里的实现也可以是简单把 DOM 结构变成使用 ngRepeat 的形式 :)

JS 部分代码：

```
1  var app = angular.module('Demo', [], angular.noop);
2
```

```
3   app.directive('for', function($compile){
4     var compile = function($element, $attrs, $link){
5       var match = $element[0].outerHTML.match('<for (.*?)=.*? in=.*?
(.*?)=.*?>');
6       if(!match || match.length != 3){throw Error('syntax: <for o in
obj_list>')}
7       var iter = match[1];
8       var list = match[2];
9       var tpl = $compile($.trim($element.html()));
10       $element.empty();
11
12       var link = function($scope, $ielement, $iattrs, $controller){
13
14       var new_node = [];
15
16       $scope.$watch(list, function(list){
17         angular.forEach(new_node, function(n){n.remove()});
18         var scp, inode;
19         for(var i = 0, ii = list.length; i < ii; i++){
20           scp = $scope.$new();
21           scp[iter] = list[i];
22           inode = tpl(scp, angular.noop);
23           $ielement.before(inode);
24           new_node.push(inode);
25         }
```

```
26
27          });
28        }
29
30        return link;
31      }
32      return {compile: compile,
33              priority: 1000,
34              terminal: true,
35              restrict: 'E'};
36  });
37
38  app.controller('TestCtrl', angular.noop);
39  angular.bootstrap(document, ['Demo']);
```

18.14. 示例：模板控制语句 if/else

这个示例是尝试实现：

```
<div ng-controller="TestCtrl">
  <if true="a == 1">
    <p>判断为真, {{ name }}</p>
  <else>
    <p>判断为假, {{ name }}</p>
  </else>
</if>
```

```
    <div>
      <p>a: <input ng-model="a" /></p>
      <p>name: <input ng-model="name" /></p>
    </div>
  </div>
```

考虑实现的思路：

- else 与 if 是两个指令，它们是父子关系。通过 scope 可以联系起来。至于 scope 是在 link 中处理还是 controller 中处理并不重要。
- true 属性的条件判断通过 $parse 服务很容易实现。
- 如果最终效果要去掉 if 节点，我们可以使用注释节点来“占位”。

JS 代码：

```
1  var app = angular.module('Demo', [], angular.noop);
2
3  app.directive('if', function($parse, $compile){
4    var compile = function($element, $attrs){
5      var cond = $parse($attrs.true);
6
7      var link = function($scope, $ielement, $iattrs, $controller){
8        $scope.if_node = $compile($.trim($ielement.html()))($scope, angular.noop);
9        $ielement.empty();
10       var mark = $('<!-- IF/ELSE -->');
11       $element.before(mark);
12       $element.remove();
13
```

```
14        $scope.$watch(function(scope){
15          if(cond(scope)){
16            mark.after($scope.if_node);
17            $scope.else_node.detach();
18          } else {
19            if($scope.else_node !== undefined){
20              mark.after($scope.else_node);
21              $scope.if_node.detach();
22            }
23          }
24        });
25      }
26        return link;
27    }
28
29    return {compile: compile,
30            scope: true,
31            restrict: 'E'}
32  });
33
34  app.directive('else', function($compile){
35    var compile = function($element, $attrs){
36
37      var link = function($scope, $ielement, $iattrs, $controller){
```

```
38          $scope.else_node = $compile($.trim($ielement.html()))($scope,
angular.noop);
39          $element.remove();
40        }
41        return link;
42      }
43
44      return {compile: compile,
45              restrict: 'E'}
46    });
47
48    app.controller('TestCtrl', function($scope){
49      $scope.a = 1;
50    });
51
52    angular.bootstrap(document, ['Demo']);
```

代码中注意一点，就是 if_node 在得到之时，就已经是做了变量绑定的了。错误的思路是，在 $watch 中再去不断地得到新的 if_node 。

原文链接：http://www.zouyesheng.com/angular.html

优化你的CSS

作者：mdemo

优化你的css，是优化你的xxx系列的第一篇，后续会推出多篇，欢迎大家来关注移动云前端小组获取最新文章。

系列概述

在移动web兴起的年代，速度优化重新被大家重视起来，因为手机的网络环境和性能比PC端差了很多，估计大家也能感觉到用手机打开网页的时候，能明显感觉到页面蜗牛般的速度。

这个系列的优化会以移动环境为基础，当然绝大多数规则也同样适合PC端。

优化的基本原则

速度优化有一些基本思路，提前总结一下

- 按需加载(只加载你需要的)
- 并行(让串行的事情并行起来)
- 压缩(通过压缩减少体积)
- 缓存(利用缓存，减少请求等待)
- 预测(预测用户行为，提前发出请求)
- 合并(把多个零散文件合并起来，减少请求)
- 自动化(让速度优化变成一种常规，和自动化工具(例如gulp,grunt,fis)结合，减少成本)

进入正题，优化CSS

为什么第一篇讲css，因为css是最难优化的，图片和js你都可以延迟加载，而css不可以，你必须在dom前面加载css，你必须接受css阻塞dom渲染的现实。

CSS优化之压缩(cssshrink)

我们一般都会对css进行常规压缩，主要做去空格和换行的工作。这里推荐的cssshrink会做更精细的工作，cssshrink会首先通过css parser对css进行解析，然后有针对性的进行优化。例如会吧0px和0%转换成0，bold转换成700，字符级别的极致压缩，为作者点个赞。

- cssshrink具体的优化策略，点此查看
- cssshrink GitHub地址

CSS优化之合并

- 使用gulp-concat将多个css合并在一起
- 不要使用@import 减少阻塞和请求

CSS拆分

看起来和上面有些冲突，这也是css和其它部分优化不同的地方。 一般我们大家都习惯把css放在最上面，js放在最下面。这是一个好习惯，但是对于css来说并不是最好的选择。

在移动端，大家非常重视首屏时间，也就是用户看到页面的时间。把整个页面的css都放在最上面，大量首屏用不到的css会阻塞首屏的展现。

- head只放首屏能用到的css，首屏外的css下移

CSS使用率

一般页面经过多人维护后，会产生大量用不到css，大家也不敢随意删除，这就需要一些检测工具

1. unu

1.1 unu是一个用来检测页面哪些css没有用到的Node.js模块

1.2 优点：提供可视化界面,使用非常简单，输入url，即可查看页面css的使用情况

1.3 缺点：目前只支持style标签式的css，另外没有执行页面的js

2. uncss

2.1 uncss是可以把页面css没有用到去除的模块

2.2 优点：支持命令行和gulp、grunt插件，支持link方式，基于phantomjs，模拟浏览器执行，支持js执行

2.3 缺点：仅凭一个url导出的css，不具有实际价值，另外不支持style标签

3. critical

3.1 critical是一个用来检测首屏css有哪些没用到的模块

3.2优点：可以输入首屏宽高来检测、有gulp、grunt插件

3.3 缺点：不支持url，只支持本地html，不支持style标签

总结

速度优化对于开发人员来说是件降低生产力的事情，所以需要尽可能的自动化，设置好规则，无痛优化，同时避免后续恶化。

原文链接：http://mweb.baidu.com/p/%E4%BC%98%E5%8C%96%E4%BD%A0%E7%9A%84css.html

Python编程中的反模式

译者：小磊

这篇文章收集了我在Python新手开发者写的代码中所见到的不规范但偶尔又很微妙的问题。本文的目的是为了帮助那些新手开发者渡过写出丑陋的Python代码的阶段。为了照顾目标读者，本文做了一些简化（例如：在讨论迭代器的时候忽略了生成器和强大的迭代工具itertools）。

对于那些新手开发者，总有一些使用反模式的理由，我已经尝试在可能的地方给出了这些理由。但通常这些反模式会造成代码缺乏可读性、更容易出bug且不符合Python的代码风格。如果你想要寻找更多的相关介绍资料，我极力推荐The Python Tutorial或Dive into Python。

迭代

- **range的使用**

Python编程新手喜欢使用range来实现简单的迭代，在迭代器的长度范围内来获取迭代器中的每一个元素：

```
for i in range(len(alist)):
    print alist[i]
```

应该牢记：range并不是为了实现序列简单的迭代。相比那些用数字定义的for循环，虽然用range实现的for循环显得很自然，但是用在序列的迭代上却容易出bug，而且不如直接构造迭代器看上去清晰：

```
for item in alist:
    print item
```

range的滥用容易造成意外的大小差一(off-by-one)错误，这通常是由于编程新手忘记了range生成的对象包括range的第一个参数而不包括第二个，

类似于java中的substring和其他众多这种类型的函数。那些认为没有超出序列结尾的编程新手将会制造出bug：

```
# 迭代整个序列错误的方法
alist = ['her', 'name', 'is', 'rio']
for i in range(0, len(alist) - 1): # 大小差一（Off by one）！
    print i, alist[i]
```

不恰当地使用range的常见理由：

1. 需要在循环中使用索引。这并不是一个合理的理由，可以用以下方式代替使用索引：

```
for index, value in enumerate(alist):
    print index, value
```

2. 需要同时迭代两个循环，用同一个索引来获取两个值。这种情况下，可以用zip来实现：

```
for word, number in zip(words, numbers):
    print word, number
```

3. 需要迭代序列的一部分。在这种情况下，仅需要迭代序列切片就可以实现，注意添加必要的注释注明用意：

```
for word in words[1:]: # 不包括第一个元素
    print word
```

有一个例外：当你迭代一个很大的序列时，切片操作引起的开销就比较大。如果序列只有10个元素，就没有什么问题；但是如果有1000万个元素时，或者在一个性能敏感的内循环中进行切片操作时，开销就变得非常重要了。这种情况下可以考虑使用xrange代替range [1]。

在用来迭代序列之外，range的一个重要用法是当你真正想要生成一个数字序列而不是用来生成索引：

```
# Print foo(x) for 0<=x<5
for x in range(5):
```

```
    print foo(x)
```

- **正确使用列表解析**

如果你有像这样的一个循环

```
# An ugly, slow way to build a list
words = ['her', 'name', 'is', 'rio']
alist = []
for word in words:
    alist.append(foo(word))
```

你可以使用列表解析来重写：

```
words = ['her', 'name', 'is', 'rio']
alist = [foo(word) for word in words]
```

为什么要这么做？一方面你避免了正确初始化列表可能带来的错误，另一方面，这样写代码让看起来很干净，整洁。对于那些有函数式编程背景的人来说，使用map函数可能感觉更熟悉，但是在我看来这种做法不太Python化。

其他的一些不使用列表解析的常见理由：

1. 需要循环嵌套。这个时候你可以嵌套整个列表解析，或者在列表解析中多行使用循环：

```
words = ['her', 'name', 'is', 'rio']
letters = []
for word in words:
    for letter in word:
        letters.append(letter)
```

使用列表解析：

words = ['her', 'name', 'is', 'rio']

letters = [letter for word in words

for letter in word]

注意：在有多个循环的列表解析中，循环有同样的顺序就像你并没有使用列表解析一样。

2. 你在循环内部需要一个条件判断。你只需要把这个条件判断添加到列表解析中去：

words = ['her', 'name', 'is', 'rio', '1', '2', '3']

alpha_words = [word for word in words if isalpha(word)]

一个不使用列表解析的合理的理由是你在列表解析里不能使用异常处理。如果迭代中一些元素可能引起异常，你需要在列表解析中通过函数调用转移可能的异常处理，或者干脆不使用列表解析。

性能缺陷

- **在线性时间内检查内容**

在语法上，检查list或者set/dict中是否包含某个元素表面上看起来没什么区别，但是表面之下却是截然不同的。如果你需要重复检查某个数据 结构里是否包含某个元素，最好使用set来代替list。（如果你想把一个值和要检查的元素联系起来，可以使用dict；这样同样可以实现常数检查时 间。

 # 假设以list开始

lyrics_list = ['her', 'name', 'is', 'rio']

 # 避免下面的写法

words = make_wordlist() # 假设返回许多要测试的单词

for word in words:

if word in lyrics_list: # 线性检查时间

```
        print word, "is in the lyrics"


# 最好这么写
lyrics_set = set(lyrics_list) # 线性时间创建set
words = make_wordlist() # 假设返回许多要测试的单词
for word in words:
    if word in lyrics_set: # 常数检查时间
        print word, "is in the lyrics"
```

[译者注：Python中set的元素和dict的键值是可哈希的，因此查找起来时间复杂度为O(1)。]

应该记住：创建set引入的是一次性开销，创建过程将花费线性时间即使成员检查花费常数时间。因此如果你需要在循环里检查成员，最好先花时间创建set，因为你只需要创建一次。

变量泄露

- 循环

通常说来，在Python中，一个变量的作用域比你在其他语言里期望的要宽。例如：在Java中下面的代码将不能通过编译：

```
// Get the index of the lowest-indexed item in the array
// that is > maxValue
for(int i = 0; i < y.length; i++) {
    if (y[i] > maxValue) {
        break;
    }
}
```

```
// i在这里出现不合法：不存在i
processArray(y, i);
```

然而在Python中，同样的代码总会顺利执行且得到意料中的结果：

```
for idx, value in enumerate(y):
    if value > max_value:
        break

processList(y, idx)
```

这段代码将会正常运行，除非子y为空的情况下，此时，循环永远不会执行，而且processList函数的调用将会抛出NameError异常，因为idx没有定义。如果你使用Pylint代码检查工具，将会警告：使用可能没有定义的变量idx。

解决办法永远是显然的，可以在循环之前设置idx为一些特殊的值，这样你就知道如果循环永远没有执行的时候你将要寻找什么。这种模式叫做哨兵模式。那么什么值可以用来作为哨兵呢？在C语言时代或者更早，当int统治编程世界的时候，对于需要返回一个期望的错误结果的函数来说为通用的模式为返回-1。例如，当你想要返回列表中某一元素的索引值：

```
def find_item(item, alist):
    # None比-1更加Python化
    result = -1
    for idx, other_item in enumerate(alist):
        if other_item == item:
            result = idx
            break
```

return result

通常情况下，在Python里None是一个比较好的哨兵值，即使它不是一贯地被Python标准类型使用（例如：str.find [2]）

- 外作用域

Python程序员新手经常喜欢把所有东西放到所谓的外作用域——python文件中不被代码块（例如函数或者类）包含的部分。外作用域相当于全局命名空间；为了这部分的讨论，你应该假设全局作用域的内容在单个Python文件的任何地方都是可以访问的。

对于定义整个模块都需要去访问的在文件顶部声明的常量，外作用域显得非常强大。给外作用域中的任何变量使用有特色的名字是明智的做法，例如，使用IN_ALL_CAPS 这个常量名。 这将不容易造成如下bug：

```
import sys

# See the bug in the function declaration?
def print_file(filenam):
    """Print every line of a file."""
    with open(filename) as input_file:
        for line in input_file:
            print line.strip()

if __name__ == "__main__":
    filename = sys.argv[1]
    print_file(filename)
```

如果你看的近一点，你将看到print_file函数的定义中用filenam命名参数名，但是函数体却引用的却是filename。然而，这个程 序仍然可以运行得很好。为什么呢？在print_file函数里，当一个局部变量filename没有被找到时，下一步是在全局作用域中去寻找。由于 print_file的调用在外作用域中(即使有缩进)，这里声明的filename对于print_file函数是可见的。

那么如何避免这样的错误呢？首先，在外作用域中不是IN_ALL_CAPS这样的全局变量就不要设置任何值[3]。参数解析最好交给main函数，因此函数中任何内部变量不在外作用域中存活。

这也提醒人们关注全局关键字global。如果你只是读取全局变量的值，你就不需要全局关键字global。你只有在想要改变全局变量名引用的对象时有使用global关键字的必要。你可以在这里获取更多相关信息this discussion of the global keyword on Stack Overflow。

代码风格

- **向PEP8致敬**

PEP 8是 Python代码的通用风格指南，你应该牢记在心并且尽可能去遵循它，尽管一些人有充分的理由不同意其中一些细小的风格，例如缩进的空格个数或使用空行。 如果你不遵循PEP8，你应该有除“我只是不喜欢那样的风格”之外更好的理由。下边的风格指南都是从PEP8中摘取的，似乎是编程者经常需要牢记的。

- **测试是否为空**

如果你要检查一个容器类型（例如：列表，词典，集合）是否为空，只需要简单测试它而不是使用类似检查len(x)>0这样的方法：

```
numbers = [-1, -2, -3]
# This will be empty
positive_numbers = [num for num in numbers if num > 0]
if positive_numbers:
    # Do something awesome
```

如果你想在其他地方保存positive_numbers是否为空的结果，可以使用bool(positive_number)作为结果保存；bool用来判断if条件判断语句的真值。

- **测试是否为None**

如前面所提到，None可以作为一个很好的哨兵值。那么如何检查它呢?

如果你明确的想要测试None，而不只是测试其他一些值为False的项（如空容器或者0），可以使用:

```
if x is not None:
    # Do something with x
```

如果你使用None作为哨兵，这也是Python风格所期望的模式，例如在你想要区分None和0的时候。

如果你只是测试变量是否为一些有用的值，一个简单的if模式通常就够用了:

```
if x:
    # Do something with x
```

例如：如果期望x是一个容器类型，但是x可能作另一个函数的返回结果值变为None，你应该立即考虑到这种情况。你需要留意是否改变了传给x的值，否则可能你认为True或0. 0是个有用的值，程序却不会按照你想要的方式执行。

译者注:

- [1] 在Python2.x 中 range生成的是list对象，xrange生成的则是range对象；Python 3.x 废除了xrange，range生成的统一为range对象，用list工厂函数可以显式生成list;
- [2] string.find(str)返回str在string中开始的索引值，如果不存在则返回-1;

- [3] 在外作用于中不要给函数中的局部变量名设置任何值，以防止函数内部调用局部变量时发生错误而调用外部作用域中的同名变量。

原译文链接：http://blog.jobbole.com/74252/

原文链接：http://lignos.org/py_antipatterns/

Web前端攻防

作者：EtherDream

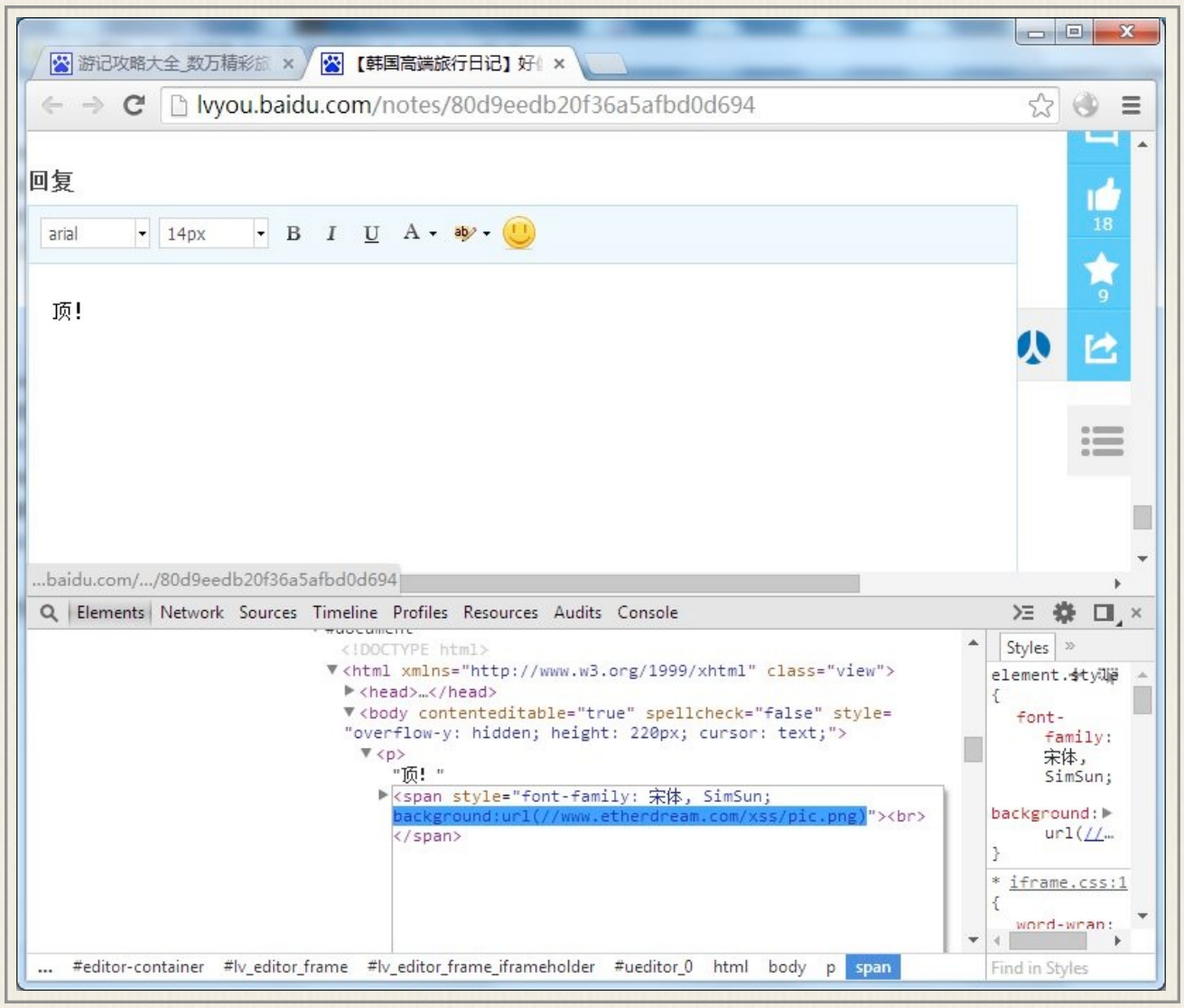
0x00 禁止一切外链资源

外链会产生站外请求，因此可以被利用实施 CSRF 攻击。

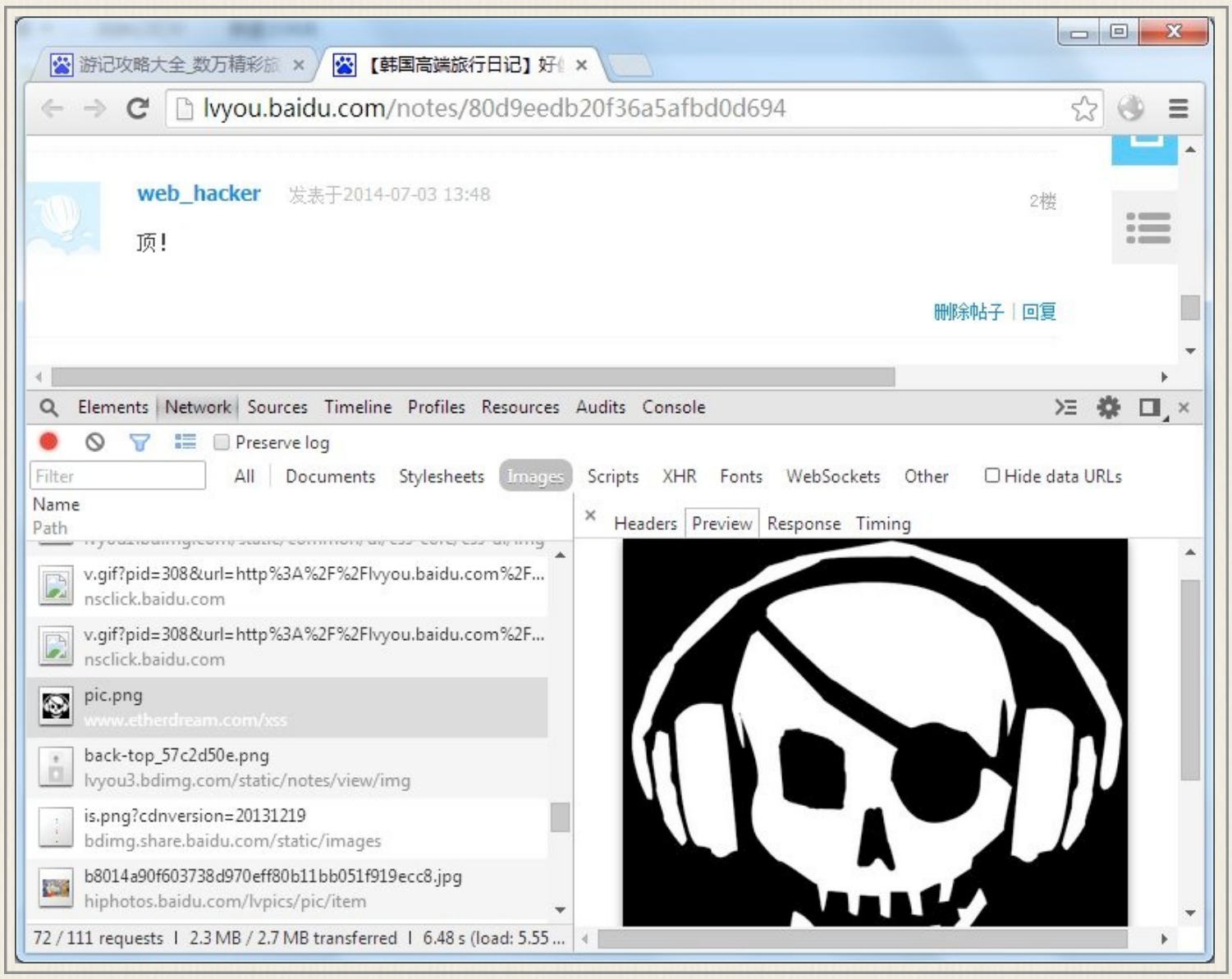
目前国内有大量路由器存在 CSRF 漏洞，其中相当部分用户使用默认的管理账号。通过外链图片，即可发起对路由器 DNS 配置的修改，这将成为国内互联网最大的安全隐患。

案例演示

百度旅游在富文本过滤时，未考虑标签的 style 属性，导致允许用户自定义的 CSS。因此可以插入站外资源：

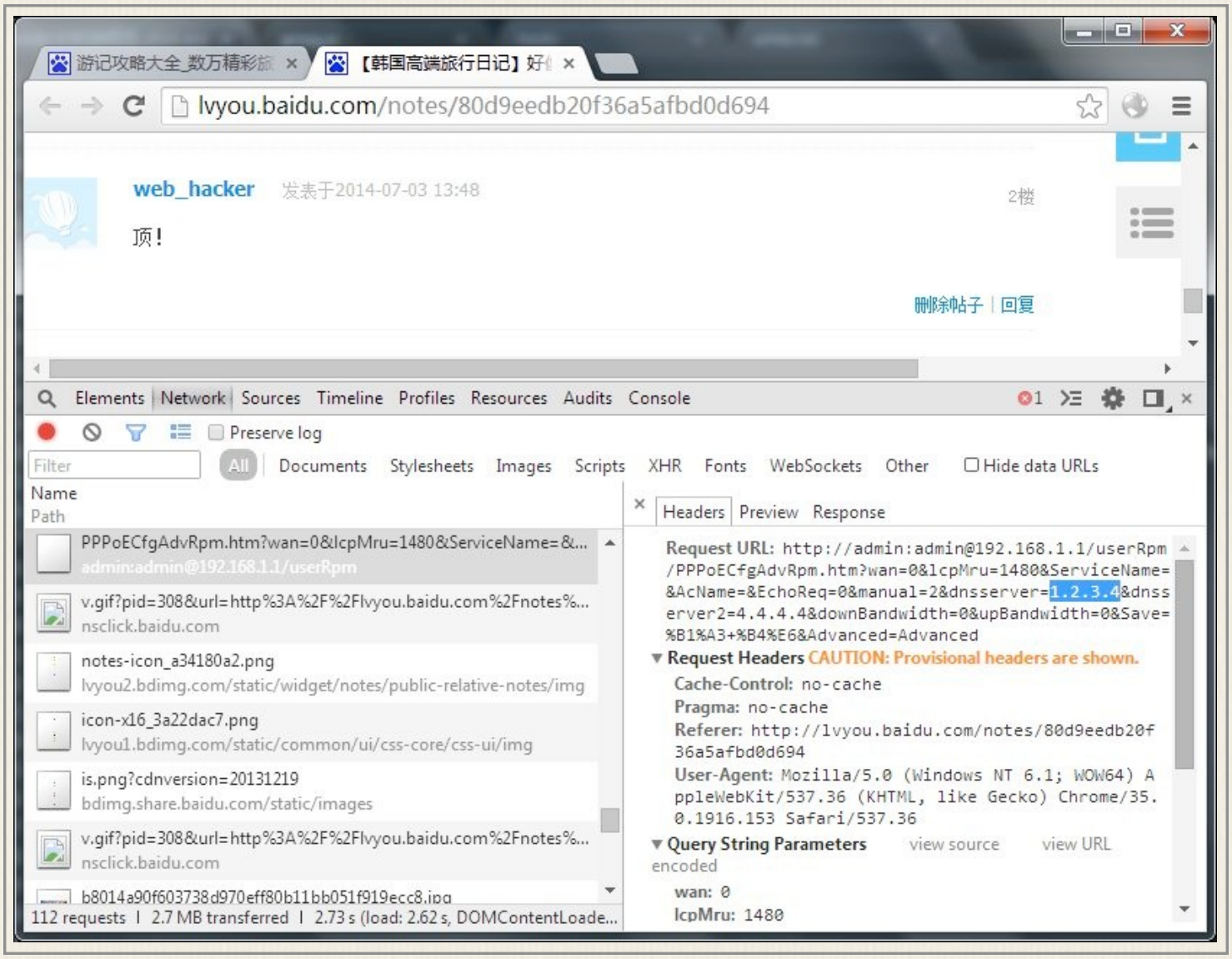


所有浏览该页面的用户，都能发起任意 URL 的请求：

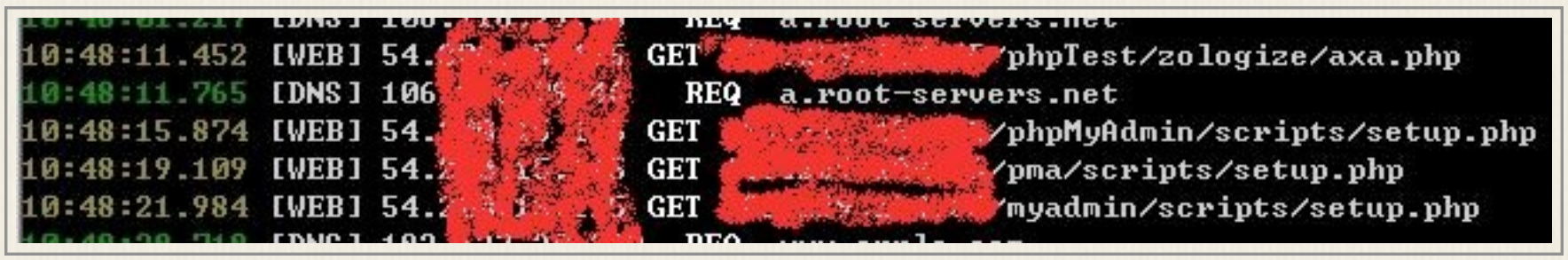


由于站外服务器完全不受控制，攻击者可以控制返回内容： 如果检测到是管理员，或者外链检查服务器，可以返回正常图片； 如果是普通用户，可以返回 302 重定向到其他 URL，发起 CSRF 攻击。例如修改路由器 DNS：

http://admin:admin@192.168.1.1/userRpm/PPPoECfgAdvRpm.htm?wan=0&lcpMru=1480&ServiceName=&AcName=&EchoReq=0&manual=2&dnsserver=黑客服务器& dnsserver2 = 4.4.4.4 & downBandwidth = 0 & upBandwidth = 0&Save=%B1%A3+%B4%E6&Advanced=Advanced



演示中，随机测试了几个帖子，在两天时间里收到图片请求 500 多次，已有近 10 个不同的 IP 开始向我们发起 DNS 查询。



通过中间人代理，用户的所有隐私都能被捕捉到。还有更严重的后果，查考流量劫持危害探讨

要是在热帖里『火前留名』，那么数量远不止这些。

如果使用发帖脚本批量回复，将有数以万计的用户网络被劫持。

防范措施

杜绝用户的一切外链资源。需要站外图片，可以抓回后保存在站内服务器里。 对于富文本内容，使用白名单策略，只允许特定的 CSS 属性。 尽可能开启 Content Security Policy 配置，让浏览器底层来实现站外资源的拦截。

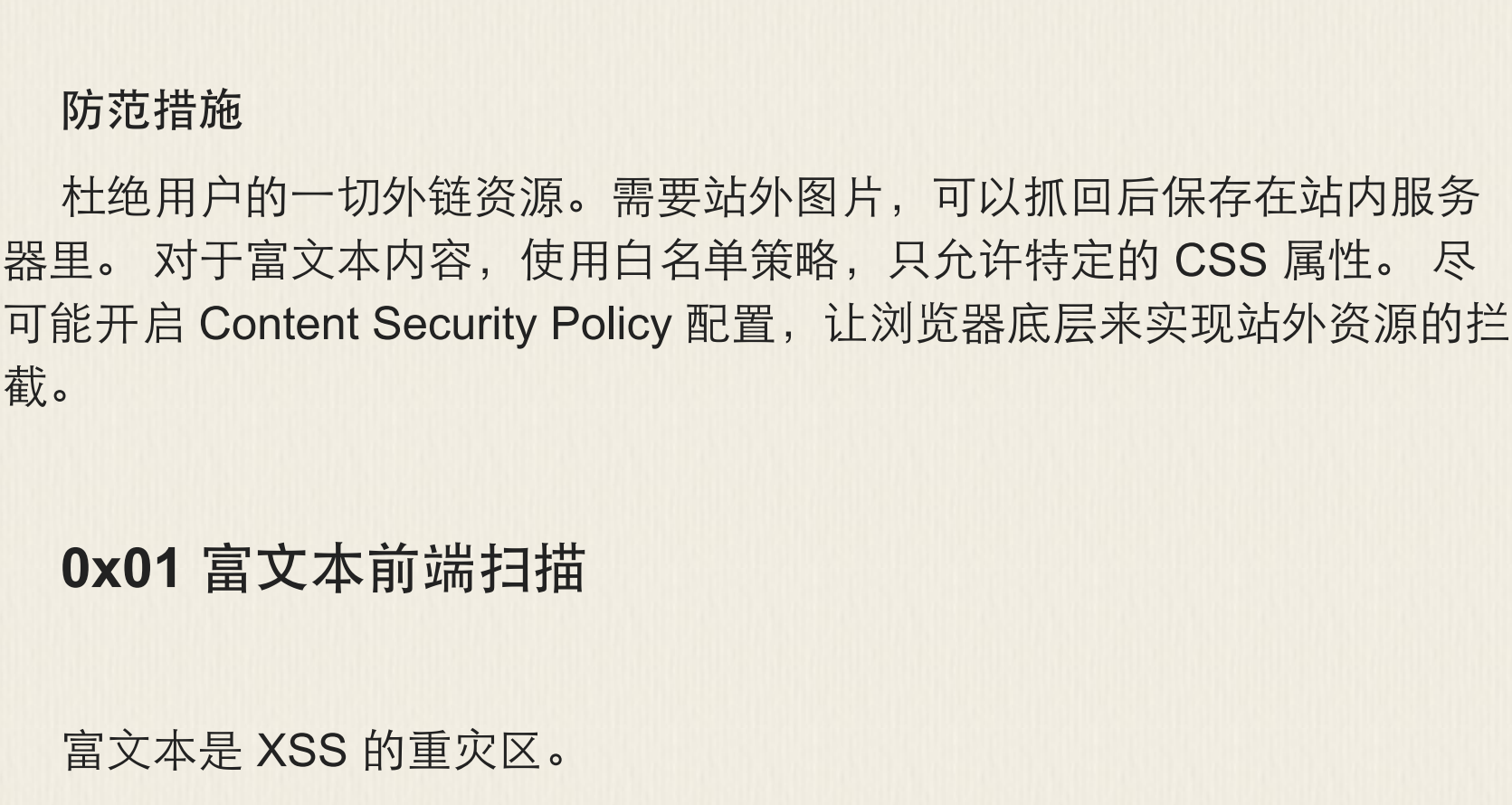
0x01 富文本前端扫描

富文本是 XSS 的重灾区。

富文本的实质是一段 HTML 字符。由于历史原因，HTML 兼容众多不规范的用法，导致过滤起来较复杂。几乎所有使用富文本的产品，都曾出现过 XSS 注入。

案例演示

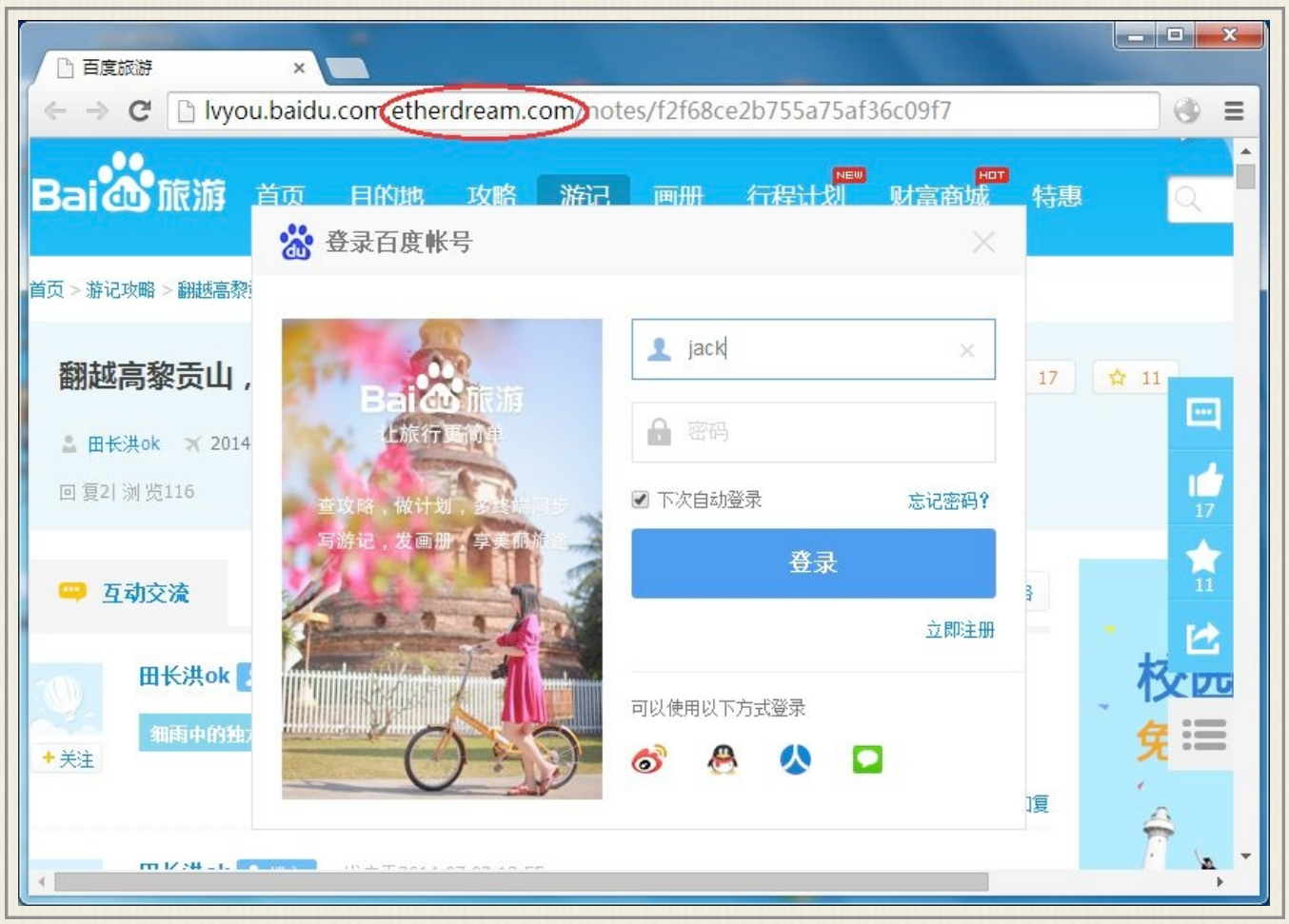
旅游发帖支持富文本，我们继续刚才的演示。



由于之前已修复过几次，目前只能注入 embed 标签和 src 属性。 但即使这样，仍然可以嵌入一个框架页面：

因为是非同源执行的 XSS，所以无法获取主页面的信息。但是可以修改 top.location，将页面跳转到第三方站点。

将原页面嵌入到全屏的 iframe 里，伪造出相同的界面。然后通过浮层登录框，进行钓鱼。



总之，富文本中出现可执行的元素，页面安全性就大打折扣了。

防范措施

这里不考虑后端的过滤方法，讲解使用前端预防方案： 无论攻击者使用各种取巧的手段，绕过后端过滤，但这些 HTML 字符最终都要在前端构造成元素，并渲染出来。

因此可以在 DOM 构造之后、渲染之前，对离屏的元素进行风险扫描。将可执行的元素（script，iframe，frame，object，embed，applet）从缓存中移除。

或者给存在风险的元素，加上沙箱隔离属性。

例如 iframe 加上 sandbox 属性，即可只显示框架内容而不运行脚本 例如 Flash 加上 allowScriptAccess 及 allowNetworking，也能起到一定的隔离作用。

DOM 仅仅被构造是不会执行的，只有添加到主节点被渲染时才会执行。所以这个过程中间，可以实施安全扫描。

实现细节可以参考：http://www.etherdream.com/FunnyScript/richtextsaferender.html

如果富文本是直接输入到静态页面里的，可以考虑使用 MutationEvent 进行防御。详细参考：http://fex.baidu.com/blog/2014/06/xss-frontend-firewall-2/

但推荐使用动态方式进行渲染，可扩展性更强，并且性能消耗最小。

0x02 跳转 opener 钓鱼

浏览器提供了一个 opener 属性，供弹出的窗口访问来源页。但该规范设计的并不合理，导致通过超链接打开的页面，也能使用 opener。

因此，用户点了网站里的超链接，导致原页面被打开的第三方页面控制。

虽然两者受到同源策略的限制，第三方无法访问原页面内容，但可以跳转原页面。

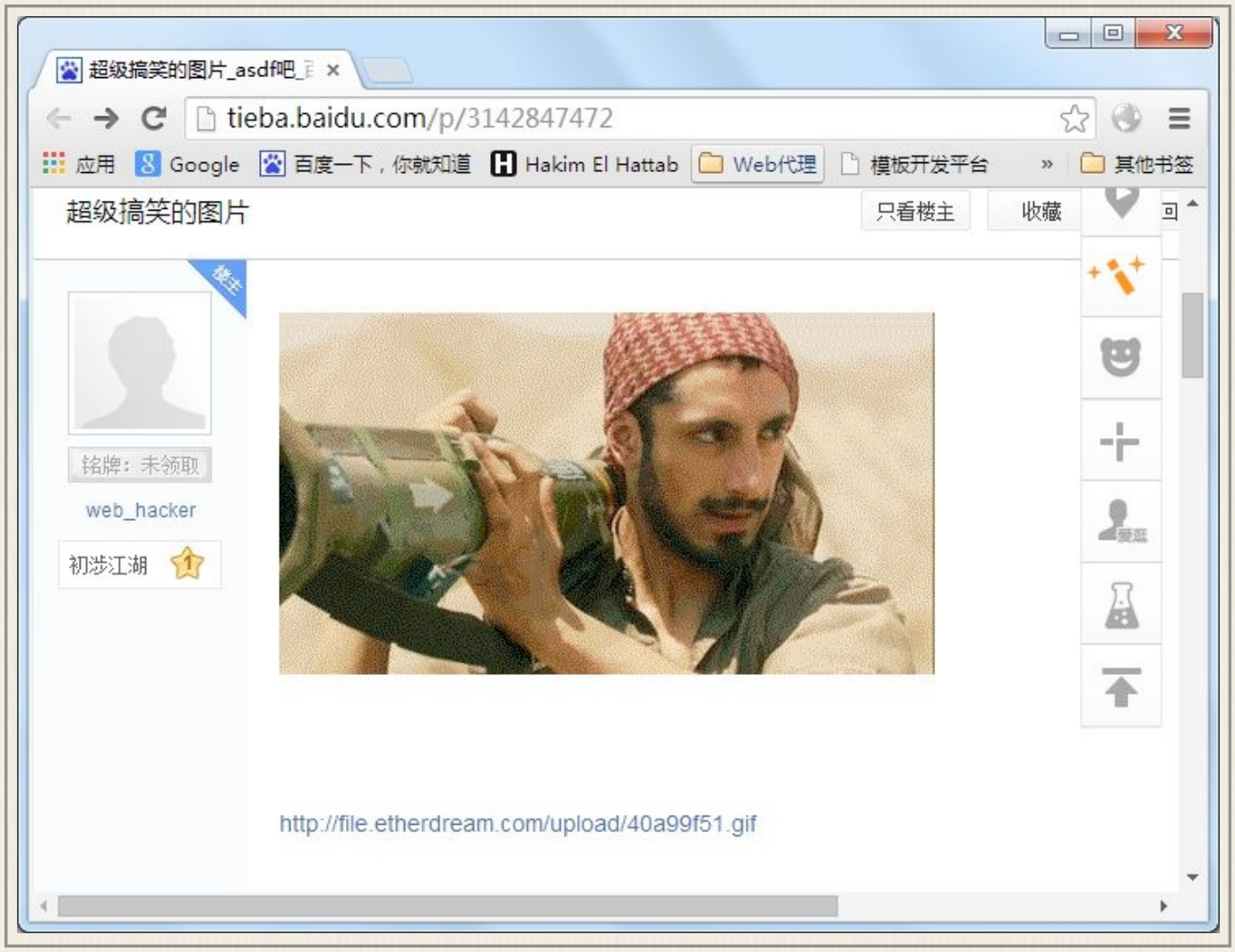
由于用户的焦点在新打开的页面上，所以原页面被悄悄跳转，用户难以觉察到。当用户切回原页面时，其实已经在另一个钓鱼网站上了。

案例演示

百度贴吧目前使用的超链接，就是在新窗口中弹出，因此同样存在该缺陷。

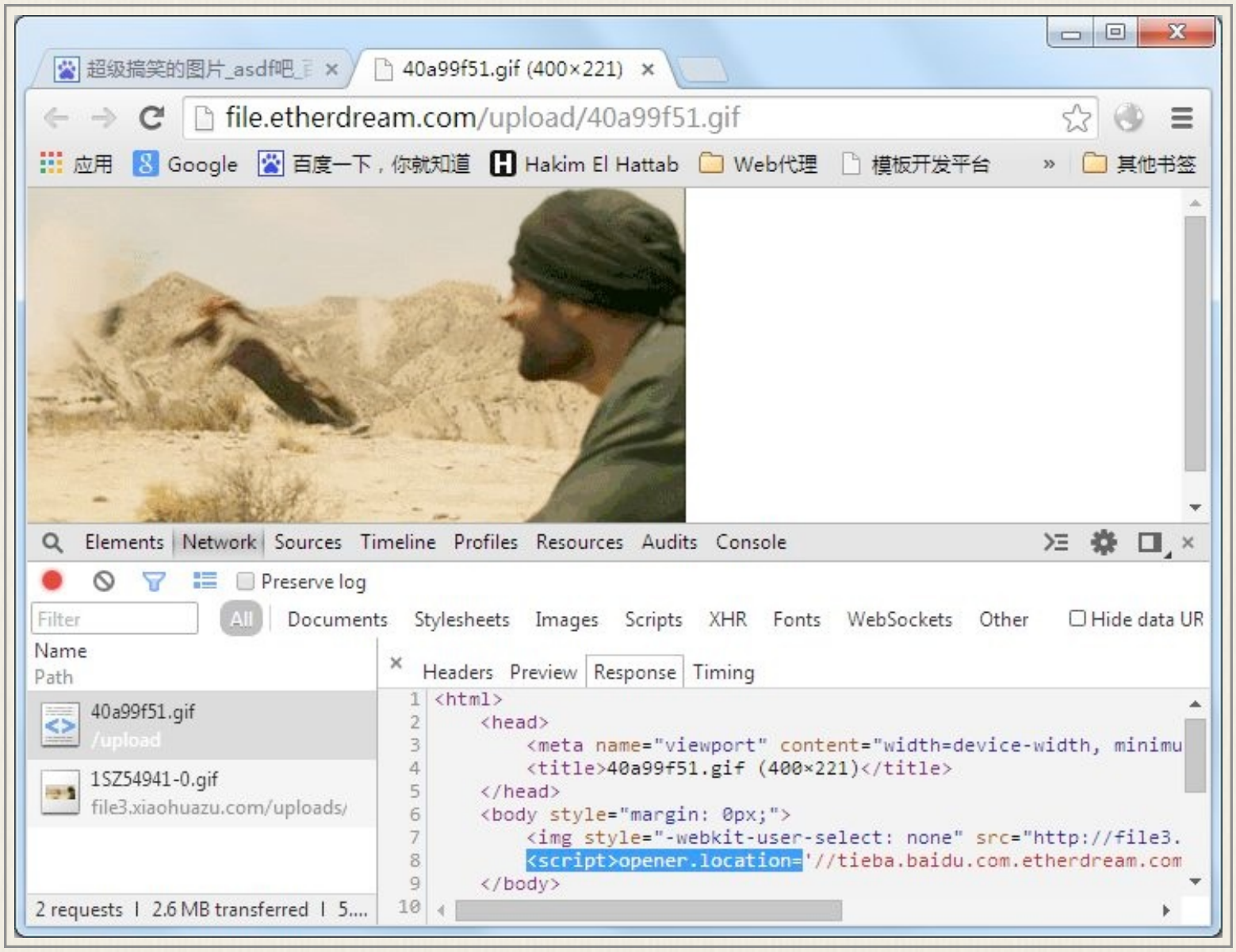
攻击者发一个吸引用户的帖子。当用户进来时，引诱他们点击超链接。

通常故意放少部分的图片，或者是不会动的动画，先让用户预览一下。要是用户想看完整的，就得点下面的超链接：



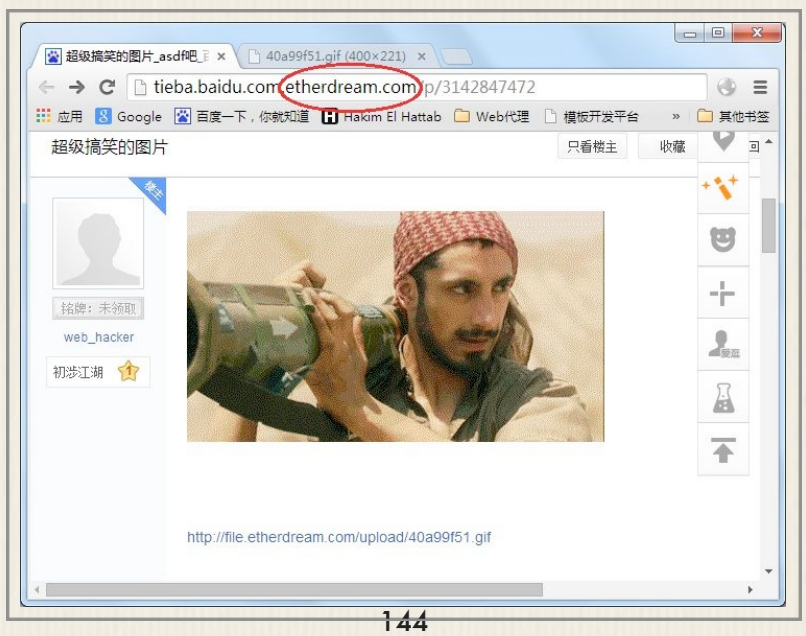
由于扩展名是 gif 等图片格式，大多用户就毫无顾虑的点了。

事实上，真正的类型是由服务器返回的 MIME 决定的。所以这个站外资源完全有可能是一个网页：



当用户停留在新页面里看动画时，隐匿其中的脚本已悄悄跳转原页面了。

用户切回原页面时，其实已在一个钓鱼网站上：



在此之上，加些浮层登录框等特效，很有可能钓到用户的一些账号信息。

防范措施

该缺陷是因为 opener 这个属性引起的，所以得屏蔽掉新页面的这个属性。

但通过超链接打开的网页，无法被脚本访问到。只有通过 window.open 弹出的窗口，才能获得其对象。

所以，对页面中的用户发布的超链接，监听其点击事件，阻止默认的弹窗行为，而是用 window.open 代替，并将返回窗体的 opener 设置为 null，即可避免第三方页面篡改了。

详细实现参考：http://www.etherdream.com/FunnyScript/opener_protect.html

当然，实现中无需上述 Demo 那样复杂。根据实际产品线，只要监听用户区域的超链接就可以。

0x03 用户内容权限

支持自定义装扮的场合，往往是钓鱼的高发区。

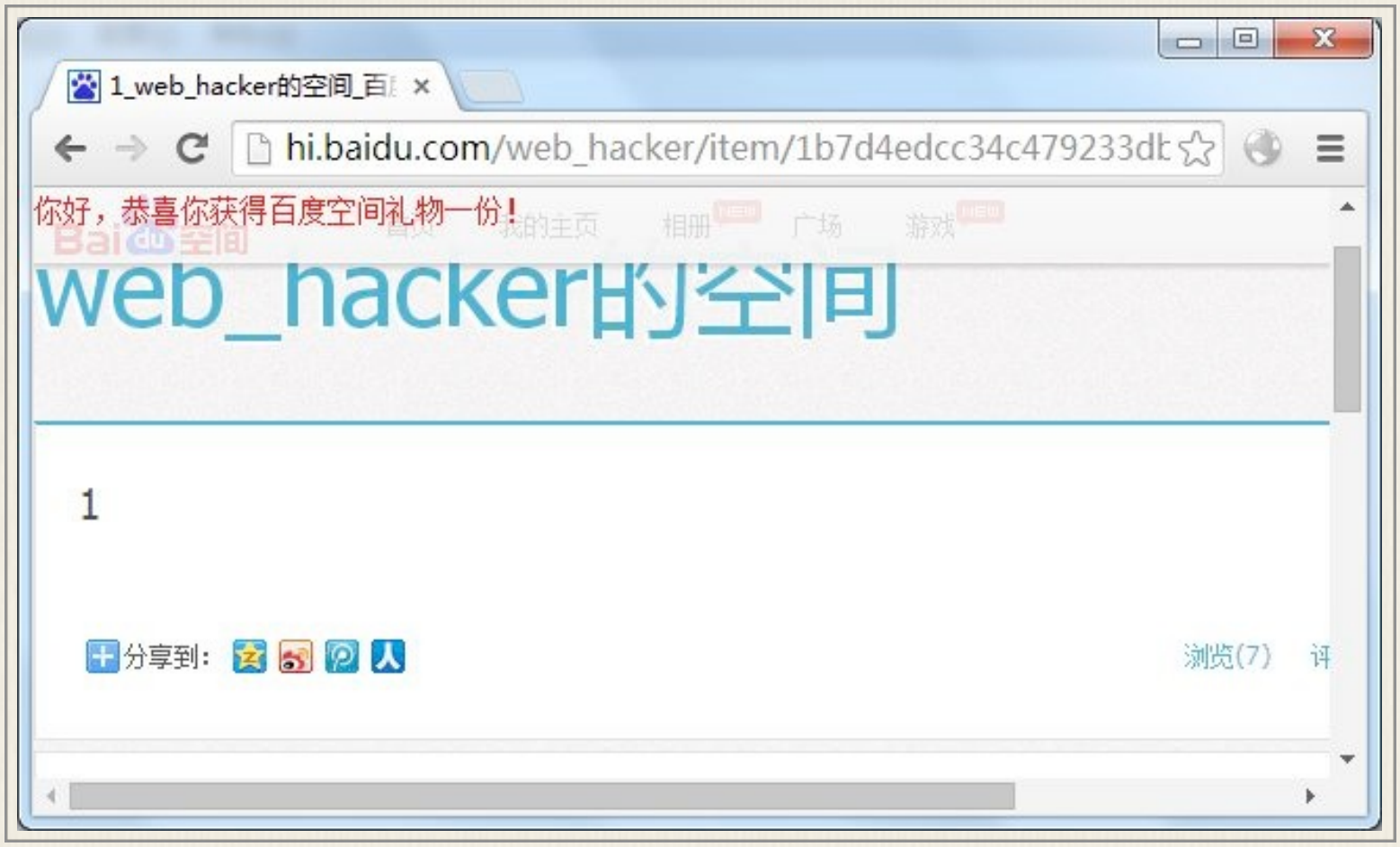
一些别有用心者，利用装扮来模仿系统界面，引诱用户上钩。

案例演示 - 空间越界

百度空间允许用户撰写自定格式的内容。

其本质是一个富文本编辑器，这里不演示 XSS 漏洞，而是利用样式装扮，伪装一个钓鱼界面。

百度空间富文本过滤元素、部分属性及 CSS 样式，但未对 class 属性启用白名单，因此可以将页面上所有的 CSS 类样式，应用到自己的内容上来。



防范措施

规定用户内容尺寸限制，可以在提交时由用户自己确定。

不应该为用户的内容分配无限的尺寸空间，以免恶意用户设置超大字体，破坏整个页面的浏览。

最好将用户自定义的内容嵌套在 iframe 里，以免影响到页面其他部位。

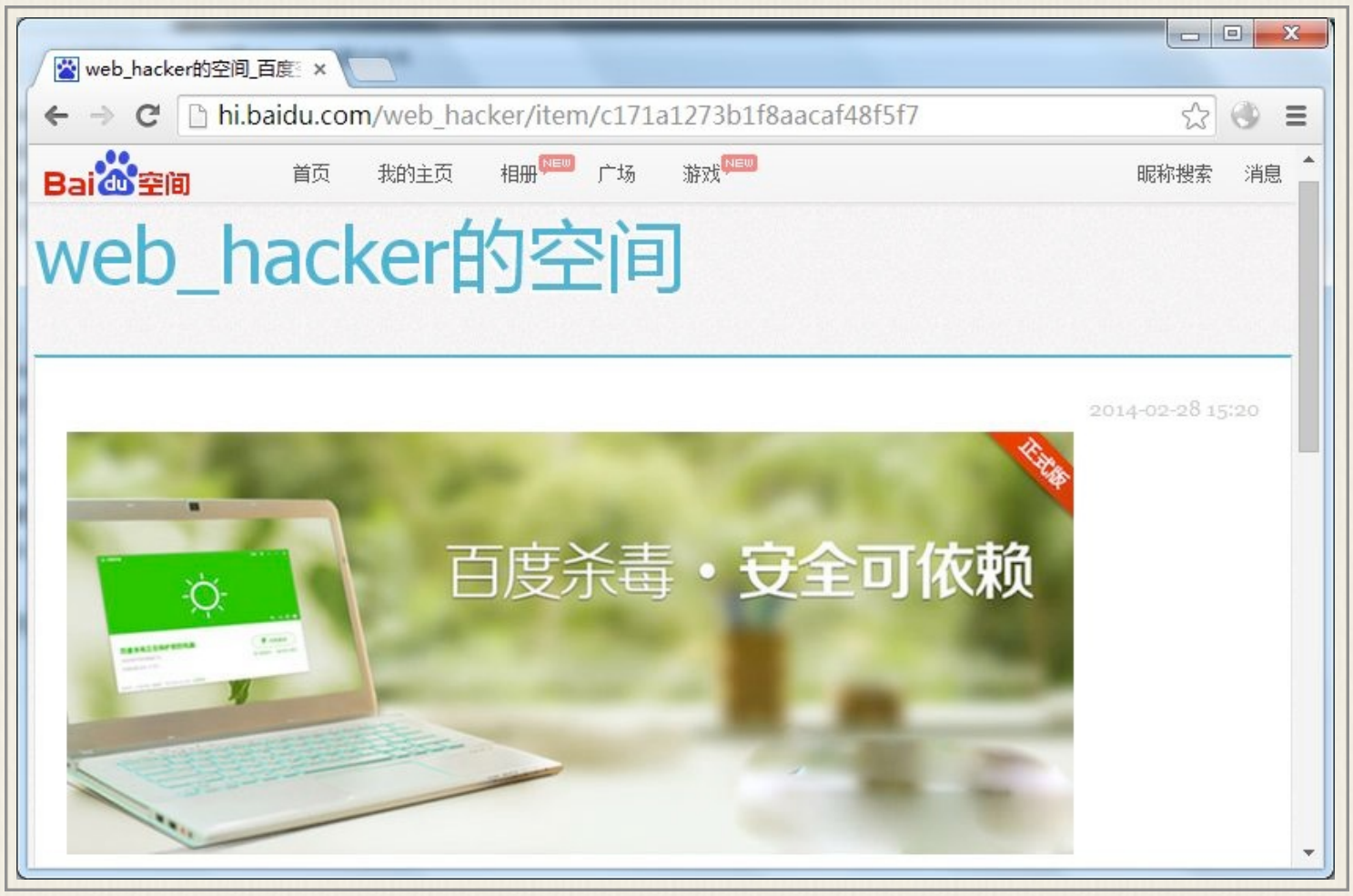
如果必须在同页面，应将用户内容所在的容器，设置超过部分不可见。以免因不可预测的 BUG，导致用户能将内容越界到产品界面上。

案例演示 - 功能越界

自定义装扮通常支持站外超链接。

相比贴吧这类简单纯文字，富文本可以将超链接设置在其他元素上，例如图片。

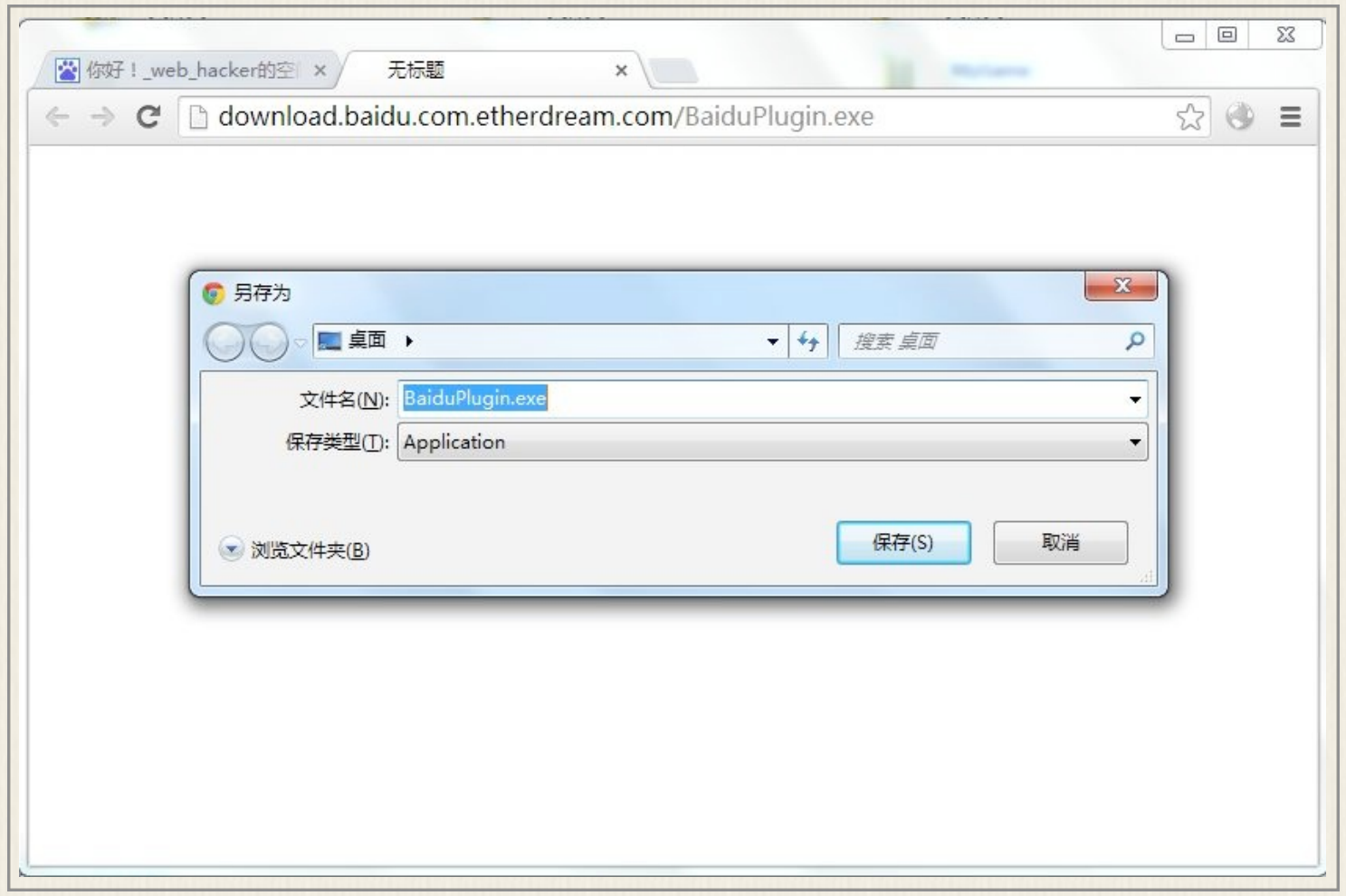
因此这类链接非常具有迷惑性，用户不经意间就点击到。很容易触发之前提到的修改 opener 钓鱼。



如果在图片内容上进行伪装，更容易让用户触发一些危险操作。

要是和之前的区域越界配合使用，迷惑性则更强：


你好！_web_hacker的空
hi.baidu.com/web_hacker/item/9238931a9e2f350df6625c41
BaiduSetUp Plugin 需要您的许可才能运行。
运行一次
在该网站上始终运行
了解详情
web_hacker的空间
你好！
2014-02-20 20:12
天气不错!
#第一篇文章
分享到:
浏览(79)
评论
编辑
转载(0)
删除
你可能也喜欢


你好！_web_hacker的空
无标题
download.baidu.com.etherdream.com/BaiduPlugin.exe
另存为
桌面
搜索 桌面
文件名(N):
BaiduPlugin.exe
保存类型(T):
Application
浏览文件夹(B)
保存(S)
取消

防范措施

和之前一样，对于用户提供的超链接，在点击时进行扫描。如果是站外地址，则通过后台跳转进入，以便后端对 URL 进行安全性扫描。

如果服务器检测到是一个恶意网站，或者目标资源是可执行文件，应给予用户强烈的警告，告知其风险。

0x04 点击劫持检测

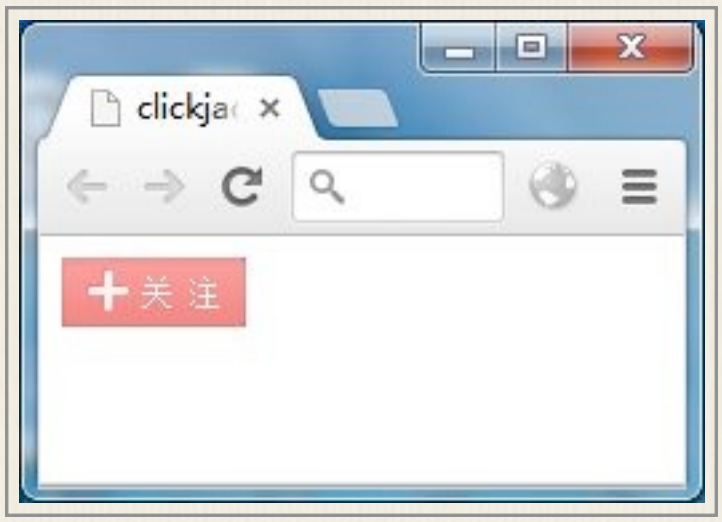
点击劫持算是比较老的攻击方式了，基本原理大家也都听说过。就是在用户不知情的前提下，点击隐藏框架页面里的按钮，触发一些重要操作。

但目前在点击劫持上做防御的并不多，包括百度绝大多数产品线目前都未考虑。

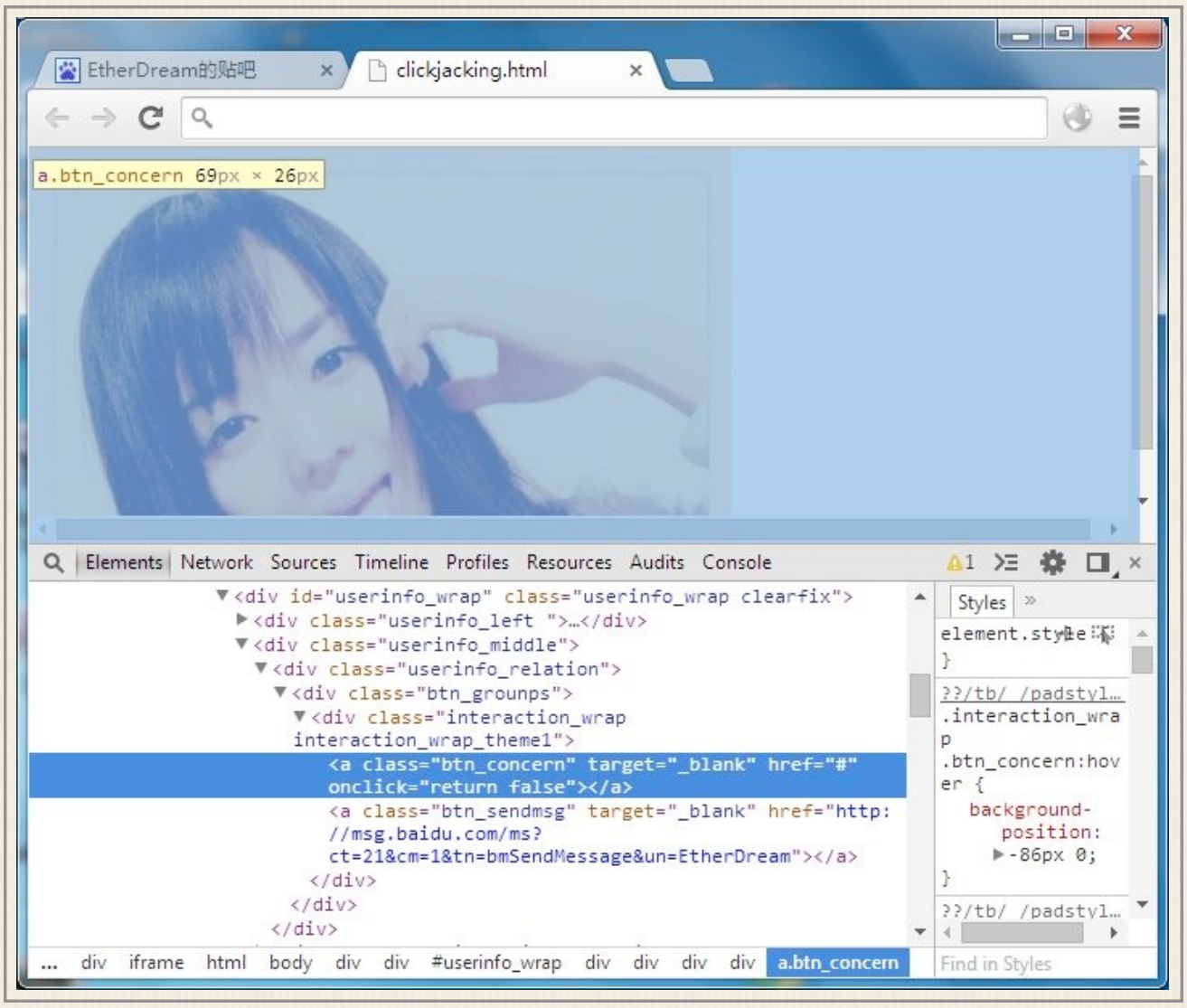
案例演示

能直接通过点击完成的操作，比较有意义的就是关注某用户。例如百度贴吧加关注的按钮：

攻击者事先算出目标按钮的尺寸和坐标，将页面嵌套在自己框架里，并设置框架的偏移，最终只显示按钮：



接着通过 CSS 样式，将目标按钮放大，占据整个页面空间，并设置全透明。



这时虽看不到按钮，但点击页面任意位置，都能触发框架页中加关注按钮的点击：

防范措施

事实上，点击劫持是很好防御的。

因为自身页面被嵌套在第三方页面里，只需判断 self == top 即可获知是否被嵌套。

对一些重要的操作，例如加关注、删帖等，应先验证是否被嵌套。如果处于第三方页面的框架里，应弹出确认框提醒用户。

确认框的坐标位置最好有一定的随机偏移，从而使攻击者构造的点击区域失效。

原文链接：http://drops.wooyun.org/tips/2686

微博推荐算法简述

作者：admin

“We are leaving the age of information and entering the age of recommendation” — Chris Anderson in The Long Tail。

在介绍微博推荐算法之前，我们先聊一聊推荐系统和推荐算法。有这样一些问题：推荐系统适用哪些场景？用来解决什么问题、具有怎样的价值？效果如何衡量？

推荐系统诞生很早，但真正被大家所重视，缘起于以”facebook”为代表的社会化网络的兴起和以“淘宝“为代表的电商的繁荣，”选择“的时代已经来临，信息和物品的极大丰富，让用户如浩瀚宇宙中的小点，无所适从。推荐系统迎来爆发的机会，变得离用户更近：

1. 快速更新的信息，使用户需要借助群体的智慧，了解当前热点。

2. 信息极度膨胀，带来了高昂的个性化信息获取成本，过滤获取有用信息的效率低下。

3. 很多情况下，用户的个性化需求很难明确表达，比如“今天晚上需要在附近找一个性价比高、又符合我口味的餐馆“。

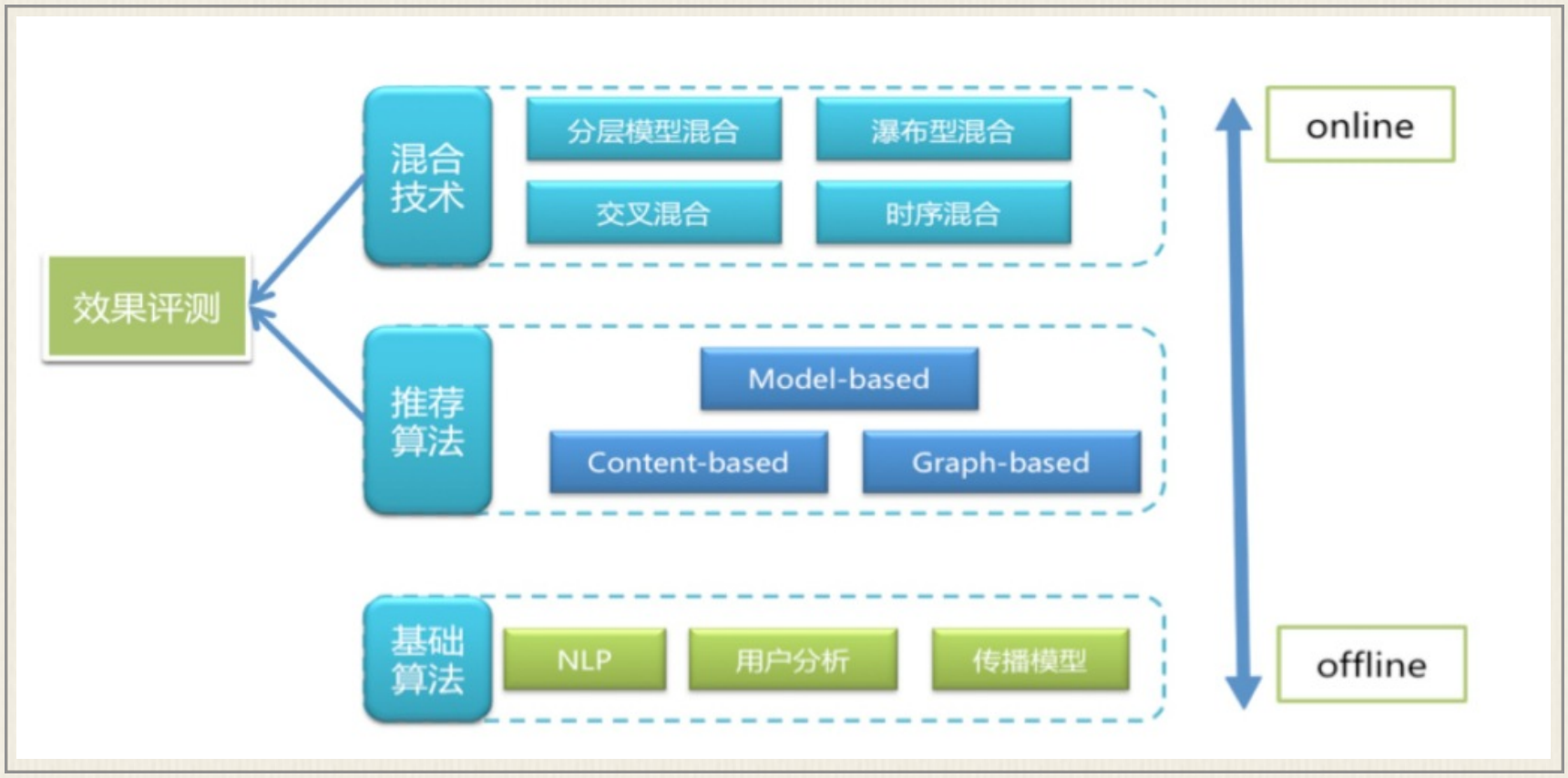
推荐系统的适用场景还有很多，不再一一列举；其主要解决的问题是为用户找到合适的item（连接和排序），并找到一个合理的理由来解释推荐结果。而问题的解决，就是系统的价值，即建立关联、促进流动和传播、加速优胜劣汰。

推荐算法是实现推荐系统目标的方法和手段。算法与产品相结合，搭载在高效稳定的架构上，才能发挥它的最大功效。

接下来我们说一下微博推荐，微博本身的产品设计，使得即使没有推荐系统，仍然会形成一个大的用户关系网络，实现信息快速传播；而衡量一个事物的价 值，一个简单的方法是对比看看保留它和去掉它时的差别。微博需要健康的用户关系网络，保障用户feed流的质量，且需要优质信息快速流动，通过传播淘汰低 质信息。微博推荐的作用在于加速这一过程，并在特定的情况下控制信息的流向，所以微博推荐的角色是一个加速器和控制器。

最后回到微博推荐算法中来，上面扯了那么多，只是为了让大家能对微博推荐算法有更好的理解。我们的工作，是将微博推荐的目标和需要解决的问题，抽样为一系列的数学问题，然后运用多种数据工具进行求解。

接下来首先用一个图梳理下我们用到的方法和技术，然后再逐一介绍。



基础及关联算法

这一层算法的主要作用是为微博推荐挖掘必要的基础资源、解决推荐时的通用技术问题、完成必要的数据分析为推荐业务提供指导。

这一部分中常用的算法和技术如下：

- 分词技术与核心词提取

是微博内容推荐的基础，用于将微博内容转化为结构化向量，包括词语切分、词语信息标注、内容核心词/实体词提取、语义依存分析等。

- 分类与anti-spam

用于微博内容推荐候选的分析，包含微博内容分类和营销广告/色情类微博识别；

内容分类采用决策树分类模型实现，共3级分类体系，148个类别；营销广告/色情类微博的识别，采用贝叶斯与最大熵的混合模型。

- 聚类技术

主要用于热点话题挖掘，以及为内容相关推荐提供关联资源。属于微博自主研发的聚类技术WVT算法（word vector topic），依据微博内容特点和传播规律设计。

- 传播模型与用户影响力分析

开展微博传播模型研究和用户网络影响力分析（包含深度影响力、广度影响力和领域内影响力）。

主要推荐算法

1. Graph-based 推荐算法

微博具有这样的特点：用户贡献内容，社会化途径传播，带来信息的爆炸式传播。之所以称作graph-based 推荐算法，而不是业界通用的memory-based 算法，主要原因在于：

- 我们的推荐算法设计是建立在社交网络之上，核心点在于从社交网络出发，融入信息传播模型，综合利用各类数据，为用户提供最佳的推荐结果；比如很多时候，我们只是信息传播的关键环节，加入必要的推荐调控，改变信息传播通路，后续的传播沿着原来的网络自然的传播。

- Feed流推荐（我们称作趋势），是我们最重要的产品，而结果必须包含用户关系。

从graph的宏观角度看，我们的目标是建立一个具有更高价值的用户关系网络，促进优质信息的快速传播，提升feed流质量；其中的重要工作是关键节点挖掘、面向关键节点的内容推荐、用户推荐。

对这部分的算法做相应的梳理，如下面的表格

算法	说明	应用举例
User-based CF	依据相似用户的群体喜好产生推荐结果	用户推荐、赞过的微博、正文页相关推荐
KeyUser-based CF	依据相似专家用户的协同过滤推荐，利用少数人的智慧；推荐的信任来自好友和社会认同	用户推荐（兴趣维度）、热点话题
Item-based CF	依据用户的历史item消费行为推荐	实时推荐、用户推荐
Edgerank	群体动态行为的快速计算	智能排序、错过的微博
Min-hash/LSH	用于海量用户关系的简化计算	用户关注相似度、粉丝相似度计算
归一化算法	Weight的归一运算，如类idf计算、分布熵，量化节点和边的价值	面向关键节点的内容推荐、用户推荐

这里的困难点在于graph的“边”怎样量化与取舍，依据多个“边”与“节点”的综合评分计算，以及与网络挖掘分析结果的融合。

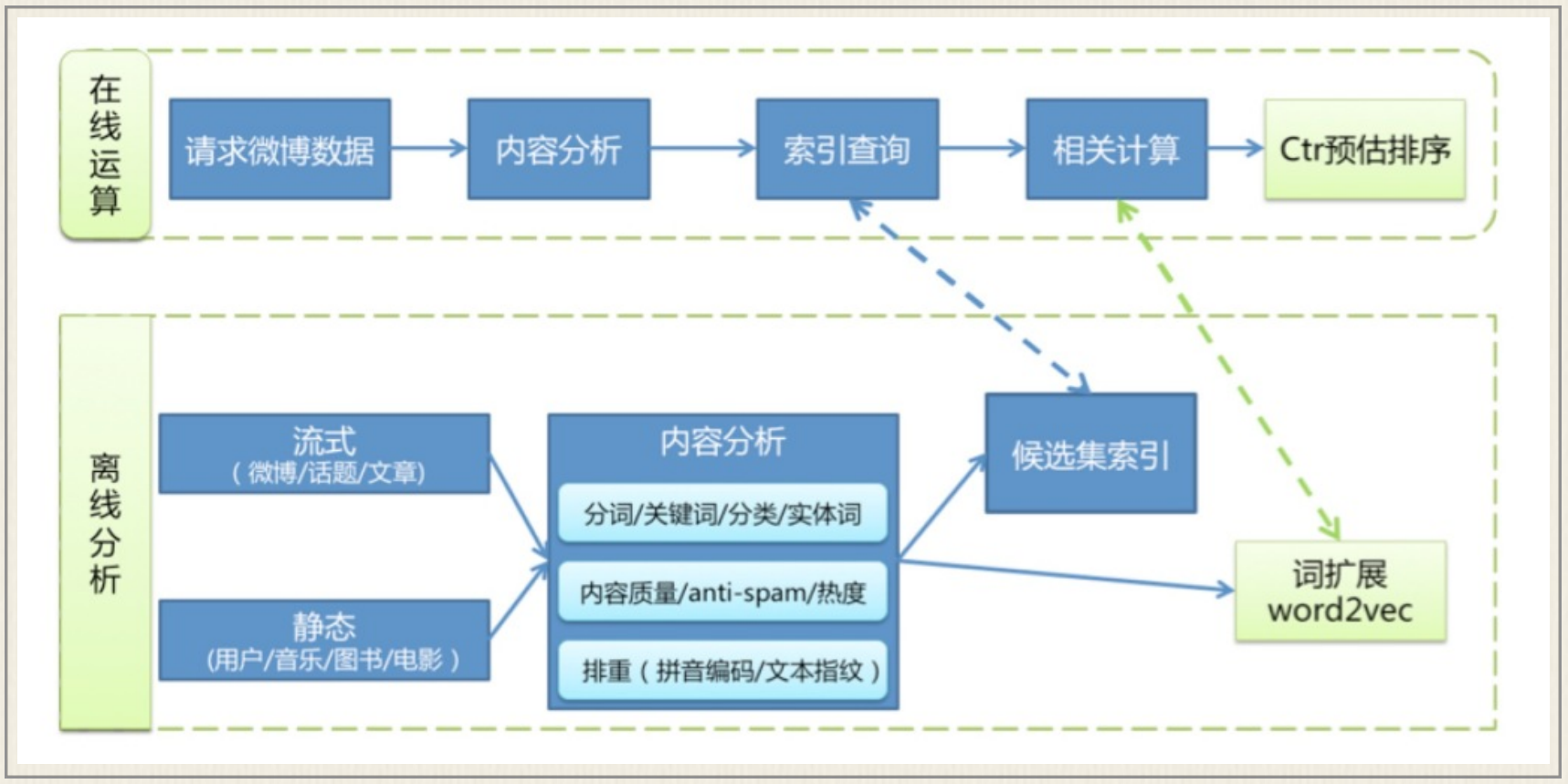
这部分的算法研发中，产出了如下的数据附产品：

数据	说明
用户亲密度	衡量user A对其follow user B的喜爱程度，是个单向分数，依据A与B的互动行为，以及A对B的主动行为计算，随着时间会逐步衰减
用户影响力	用户在微博信息传播过程中的社会化影响力，分广度影响力、深度影响力、领域影响力
关注相似度	为用户计算与其关注口味相似的用户列表，是user-based CF的基础资源
粉丝相似度	为用户计算与其具有粉丝相似的用户列表，应用于用户推荐的实时反馈
关键节点	影响信息传播的关键用户，以及具有连续优质内容生产能力的用户。通过节点信息的传播效率来计算。
兴趣协同用户	采用LDA模型对用户关系网络进行聚类分析，挖掘得到相同兴趣能力的用户。

2. Content-based 推荐算法

Content-based 是微博推荐中最常用也是最基础的推荐算法，它的主要技术环节在于候选集的内容结构化分析和相关性运算。

正文页相关推荐是content-based 应用最广的地方，以它为例，简要的说一下



内容分析的很多点已在前面描述过了，这里重点说2个地方：

- 内容质量分析，主要采用微博曝光收益+内容信息量/可读性的方法来综合计算。微博曝光收益是借助用户群体行为，衡量内容优劣；内容信息量计算比较 简单，即是微博关键词的idf信息迭代；对于内容可读性的衡量，我们做了一个小的分类模型，分别以可读性较好的新闻语料和可读性较差的口语化语料为训练样 本，通过提取里面的各类词搭配信息，计算新微博具有良好可读性的概率。

- 词扩展，content-based的效果取决于内容分析的深度。微博的内容比较短，可提取的关键信息比较少，做相关运算时容易因为数据稀疏而导致推荐召回率和准确率的难以权衡；我们引入word2vec技术，优化了词扩

展效果，后面又以此为基础开展词聚类的工作，实现了推荐召回率和准确率的同步 提升。

相关计算的技术点在于向量的量化和距离度量，我们通常使用“tf*id-f权重量化 + 余弦距离”或者“topic 概率 + KLD距离“的两种方法。

3. Model-based 推荐算法

微博作为中国最大的社会化媒体产品，具有海量的用户和信息资源；这就给推荐带来了2个挑战：

- 来源融合与排序

候选的极大丰富，意味着我们有更多的选择，于是我们推荐结果的产生包含两层：多种推荐算法的初选与来源融合排序的精选，为了得到更客观准确的排序结果，我们需要引入机器学习模型，来学习隐藏在用户群体行为背后的规律。

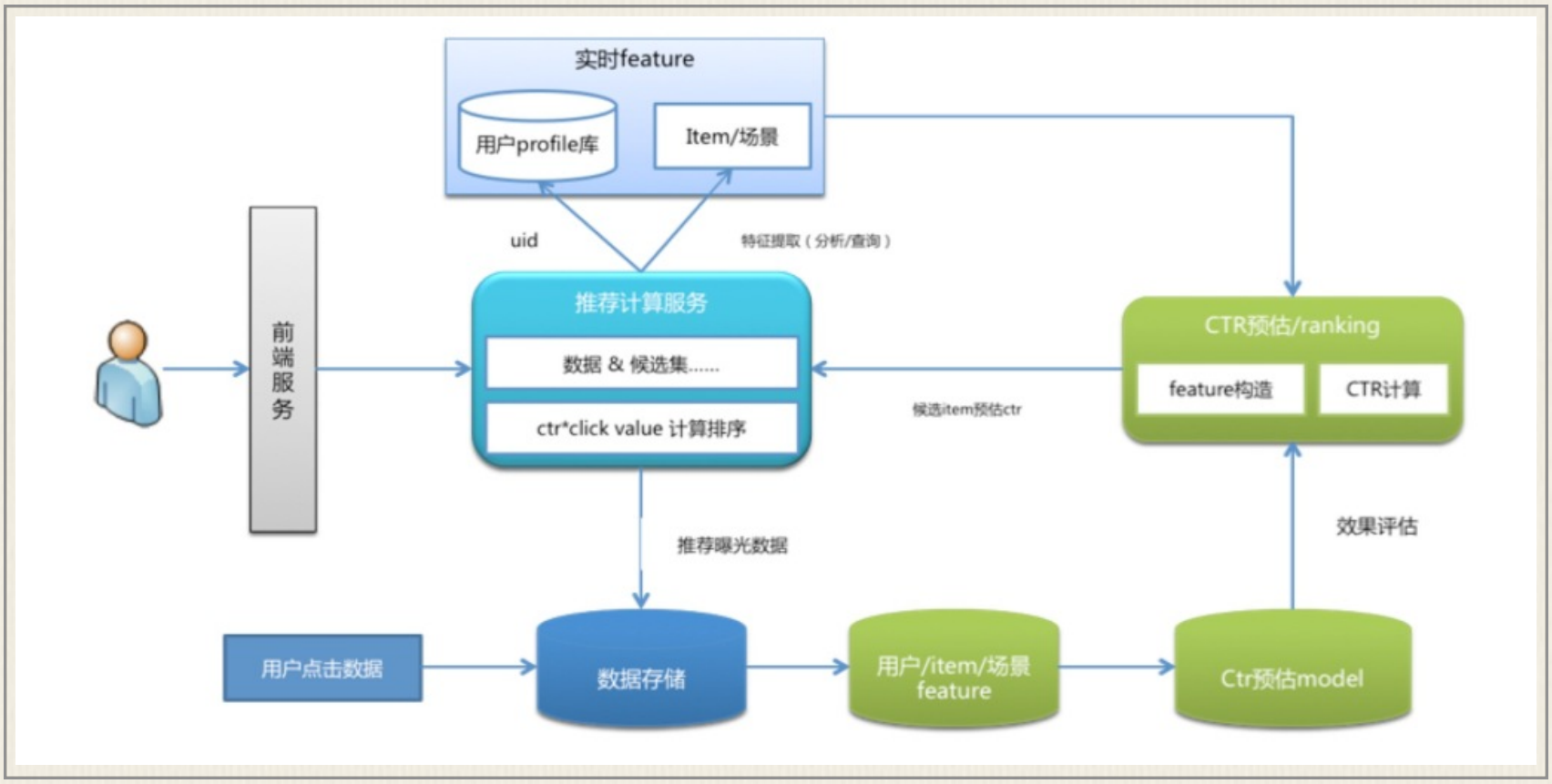
- 内容动态分类和语义相关

微博UGC的内容生产模式，以及信息快速传播和更新的特点，意味着之前人工标注样本，训练静态分类模型的方法已经过时了，我们需要很好的聚类模型把近期的全量信息聚合成类，然后建立语义相关，完成推荐。

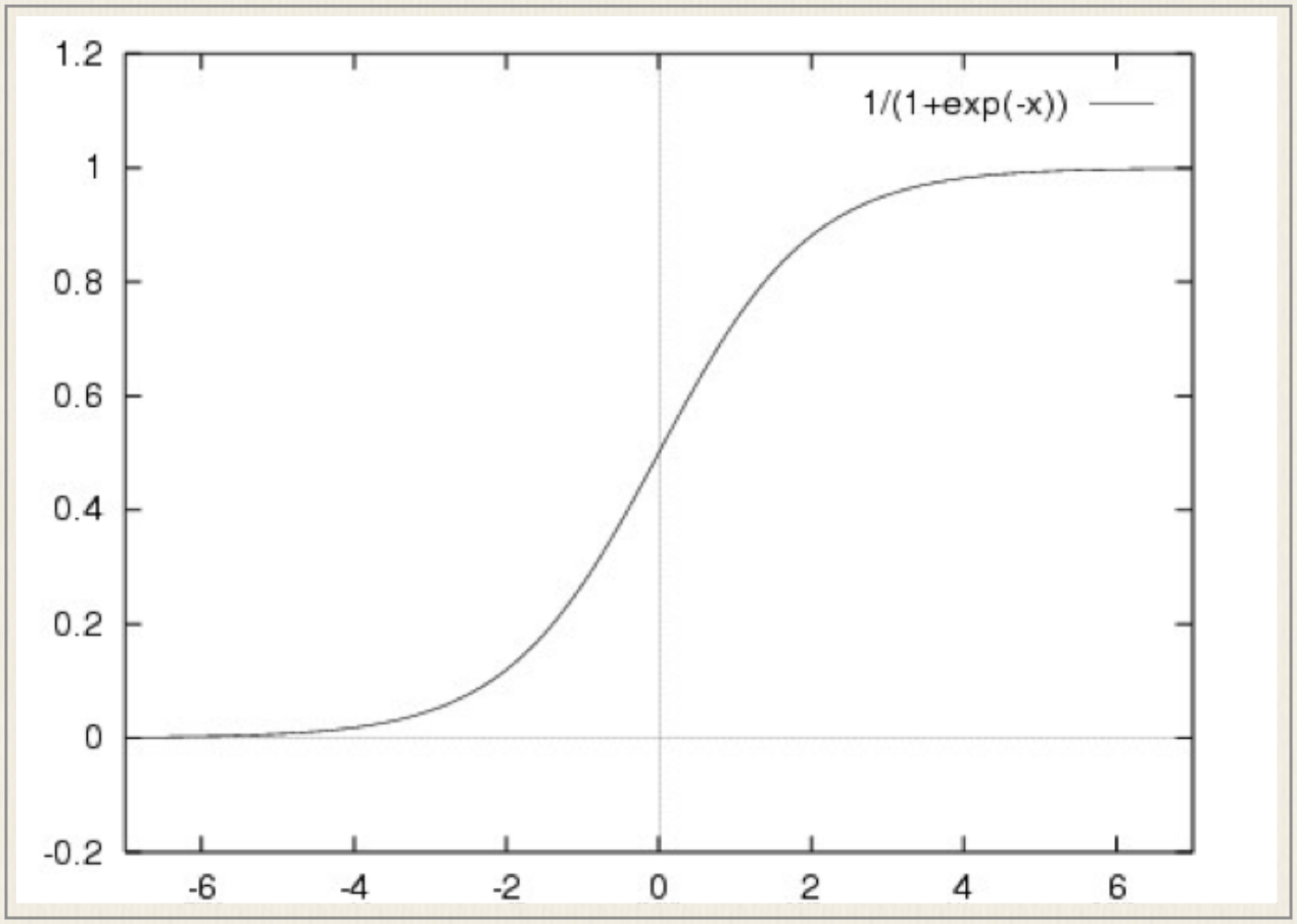
Model-based 算法就是为了解决上述的问题，下面是我们两块最重要的机器学习工作：

3.1 CTR/RPM （每千次推荐关系达成率）预估模型，采用的基本算法为Logistic regression，下面是我们CTR预估模型整体的架构图

这部分工作包含样本选择、数据清洗、特征提取与选择、模型训练、在线预估和排序。值得一提的是，模型训练前的数据清洗和噪音剔除非常重要，数据质量是算法效果的上界，我们之前就在这个地方吃过亏。



Logisitic regression是一个2分类概率模型



3.2 LFM（Latent Factor Model）：LDA、矩阵分解（SVD++、SVD Feature）

LDA是2014年初重点开展的项目，现在已经有了较好的产出，也在推荐线上产品中得到了应用；LDA本身是一个非常漂亮和严谨的数学模型，下面是我们一个LDA topic的例子，仅供参考。

至于矩阵分解，2013年的时候做过相应的尝试，效果不是特别理想，没有继续投入。

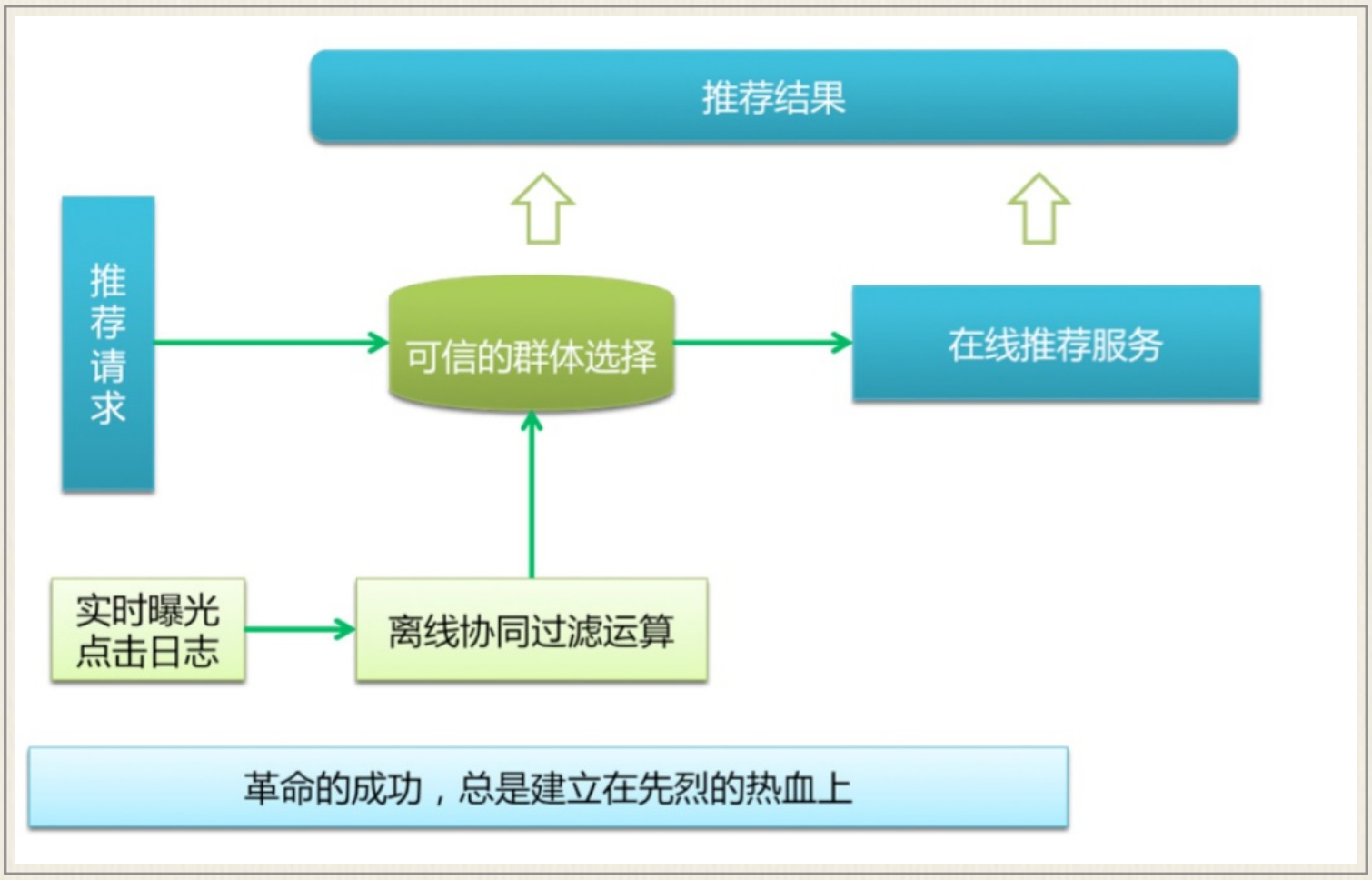
隐语义模型是推荐精度最高的单一模型，其困难在于数据规模大时，计算效率会成为瓶颈；我们在这个地方开展了一些工作，后续会有同学专门介绍这一块。

混合技术

三个臭皮匠顶个诸葛亮，每一种方法都有其局限性，将不同的算法取长补短，各自发挥价值，是极为有效的方式。微博推荐算法主要采用了下面的混合技术：

- 时序混合：

即在推荐过程的不同时间段，采用不同的推荐算法；以正文页相关推荐为例，在正文页曝光的前期阶段，采用content-based + ctr预估的方法生成推荐结果，待产生的足量可信的用户点击行为后，再采用user-based 协同过滤的方法得到推荐结果，如下图所示：



这样利用content-based很好的解决了冷启动的问题，又充分发挥了user-based CF的作用，实现1+1>2的效果。

- 分层模型混合：

很多情况下，一个模型无法很好的得到想要的效果，而分层组合往往会取得比较好的效果，分层模型混合即“将上一层模型的输出作为下层模型的特征值，来 综合训练模型，完成推荐任务“。比如我们在做微博首页右侧的ctr预估排序时，采用分层逻辑回归模型，解决了不同产品间特征天然缺失与样本量差异、曝光位 置带来的效果偏差等问题。

- 瀑布型混合：

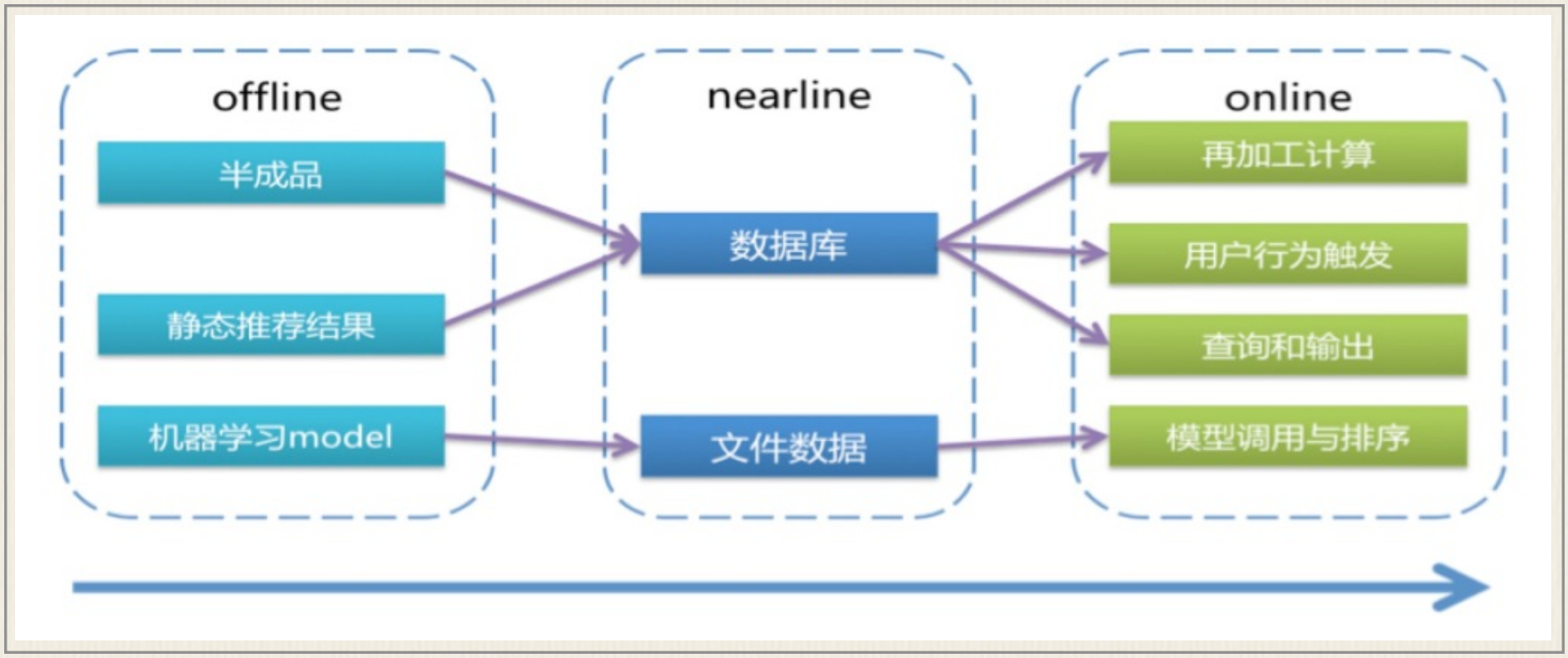
这类混合技术思路非常简单，即在推荐候选非常丰富的情况下，采用逐层过滤的方法的得到推荐结果，通常将运算快、区分度低的算法放在前面，完成大量候 选集的筛选；将运算慢、区分度高的算法放在后面，精细计算剩下的小规模集合。这类混合在微博推荐中大量使用，我们采用各种轻量算法完成候选集粗选，然后采 用ctr预估做精细化排序。

- 交叉混合：

各类推荐算法中子技术，可以在另外的推荐算法中综合使用，比如content-based在相关性计算中积累的距离计算方法，可以很好的应用在协同过滤的量化计算中。实际的例子，我们将研究LDA时积累的向量计算方法成功的应用到用户推荐中。

Online 与 offline

微博数据的特点（海量、多样、静态与动态数据混在一起），决定了大部分推荐产品的结果需要同时借助online和offline的计算来完成。从系 统和算法设计的角度，这是一个“重”与“轻”的问题，计算分解和组合是关键，我们需要将对时间不敏感的重型计算放在offline端，而将时间敏感性强的 轻型快速计算放在online端。几种我们常用的方式如下图：



Onli-
ne需要简单可靠的算法，快速得到结果；简要说明下上面的图，如下

- 半成品有以下的3中形式

1）计算过程拆解的离线部分，如user-based CF中的用户相似度，online通过数据库读取后在线计算完成user-based 推荐。

2）离线挖掘的优质候选集，如正文页相关推荐的内容候选集，online通过索引获取到数据后，再通过相关性和ctr预估排序生成推荐结果。

3）具有较高相似度的推荐结果集，如offline计算好粉丝相似高的用户，在线对用户行为做出实时反馈，实时补充推荐与其刚关注用户相似的用户。

- 静态推荐结果，是指那些与时间关联小的推荐item，如我们的用户推荐95%的结果来自离线计算。

- 机器学习模型，这是一个计算过程时序性上的拆解；offline完成模型的训练，在线调用model完成item排序，当然也可以通过online-learning或实时特征值完成模型的实时更新。同时，model在线计算时，需要注意缺失特征值的补全，保证offline与 online环境的一致性。

此外，我们也有直接online计算完成的推荐结果，如首页右侧话题推荐，由于用户对话题需求的差异非常小，它基本上是一个排行榜的需求，但热门

微 博也可以有精巧的设计，我们采用了一个曝光动态收益模型，通过上一段时段的（点击收益-曝光成本）来控制下一时段的item曝光几率，取得了非常好的效 果，ctr和导流量有3倍以上的提升。

不同类型的推荐结果，要辅以不同的推荐理由，这一点需要前端的多种展示尝试和offline的日志分析。

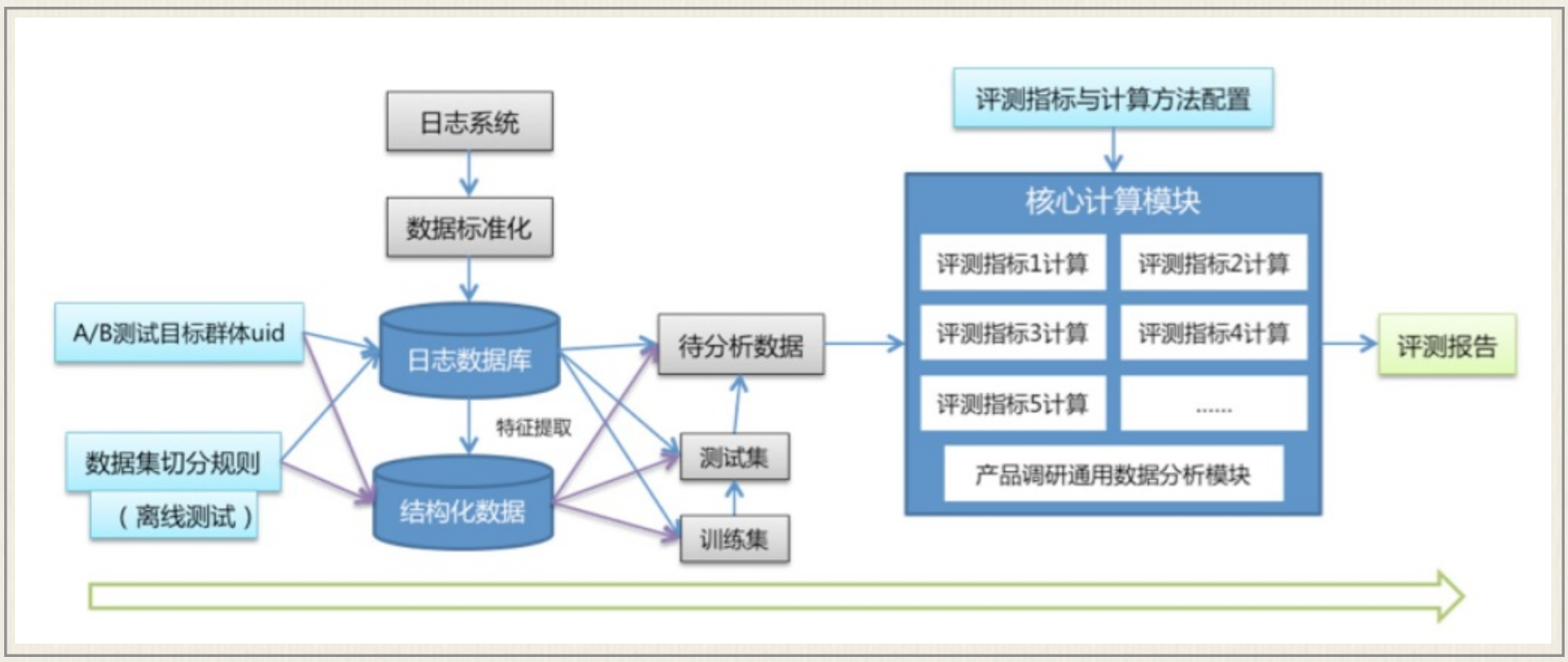
效果评测

算法效果的度量方式决定了大家努力的方向，而对于不同类型的推荐，最好根据产品的定位和目标，采用不同的标准体系去衡量工作结果。实际效果的评测分 为3个层次：用户满意度、产品层指标（如ctr）、算法层指标，我们的效果评测也会分为人工评测、线上A/B测试、离线算法效果评测3种。

产品指标的制定，应该从产品期望达成的目标出发，体现用户满意度。

对算法离线评测而言，关键的是找到一套合理的算法评测指标去拟合产品层指标，因为算法离线评测总是在上线前进行，这个对应做的越好，算法的优化成果才能更好的转化为线上的产品指标。

下图为我们的算法离线效果评测的架构图



常用的离线评测指标有：RMSE、召回率、AUC、用户内多样性、用户间多样性、新颖性等。对于不同的产品有不同的组合指标去衡量，比如用户推荐中“用户间多样性”非常重要，而热点话题却可以允许用户间有较大的结果重合度。

原文链接：http://www.wbrecom.com/?p=80

Andrew Ng谈Deep Learning

作者：卢鹓翔

摘要：Andrew Ng是Deep Learning领域最富盛名的科学家之一，他也是斯坦福人工智能实验室主任，在线教育平台Coursera的创始人之一。2014年5月，他加入百度，担任首席科学家。日前，他接受了《程序员》的专访。

Deep Learning与AI

《程序员》：Amara法则说，“我们倾向于高估科技的短期影响力，而又低估其长期影响力。”在你看来，Deep Learning的短期和长期影响分别是什么？历史上，我们曾对实现人工智能有过错误估计，对于Deep Learning的前景，人们是否过于乐观？

Andrew Ng：我对Deep Learning的前景很乐观，它的价值在过去几年已得到印证，未来我们还会沿着这个方向继续努力。语音识别、计算机视觉都将获得长足进步，数据与科技的 碰撞，会让这一切变得更具价值。在短期，我们会看到身边的产品变得更好；而长期，它有潜力改变我们与计算机的交互方式，并凭借它创造新的产品和服务。

不过围绕Deep Learning，我也看到存在着某种程度的夸大，这是一种不健康的氛围。它不单出现于媒体的字里行间，也存在于一些研究者之中。将Deep Learning描绘成对人脑的模拟，这种说法颇具吸引力，但却是过于简化的模仿，它距离真正的AI或人们所谓的“奇点”还相当遥远。目前这项技术主要是 从海量数据当中学习，理解数据，这也是现今有关Deep Learning技术研究和产品发展的驱动力。而具备与人的能力相匹配的AI需要无所不包，例如人类拥有丰富的感情、不同的动机，以及同感能力。这些都是 当下Deep Learning研究尚未涉及的。

《程序员》：关于神经网络的研究，在很多方面依靠生物学、神经科学等领域。在你看来，Deep Learning 的模型是否已经完善？若没有，目前最大的缺陷或困难在于何处?

Andrew Ng：Deep Learning模型尚未完善，主要存在两项挑战。一项是可扩展性，我们坐拥海量数据，却难以建造计算能力足够强大的计算机系统，处理这些数据。我青睐百 度的原因之一，即在于它拥有复杂而强大的海量数据处理基础架构，但对Deep Learning来说，问题尚未得到解决。另一项挑战在于算法，我们也不知道恰当的算法是什么。从这两点看，尽管我们已取得了不小的进步，但前路依然漫 漫。

《程序员》：为了开发智能机器，许多年前，Daniel Hillis和他的Thinking Machines曾尝试突破von Neumann架构，你觉得当今的硬件是否是实现智能机器的最好选择？如果不是，当前的计算机架构有哪些限制，我们需要做哪方面的突破?

Andrew Ng：这是一个有趣的话题。我们尚不知道怎样的硬件架构是智能机器的最佳选择，因而更需要拥有灵活性，快速尝试不同的算法。在这方面，GPU相对易于编 程，因而可以高效地尝试不同的算法。作为对比，ASIC（专用集成电路）的运行速度比GPU更快，但开发适合Deep Learning的ASIC 难度高、周期长，在漫长的研发过程中，很可能我们早已发现了新算法。

GPU与CPU结合是目前的首选硬件平台，不过随着技术的进一步成熟，这种现状有可能发生改观。例如，目前已有几家初创公司正在研发专门用于Deep Learning的硬件系统。

《程序员》：有一种说法是，对人脑机制理解的缺乏是我们开发智能机器的最大限制之一，在这个存在许多假设和未知的前沿领域进行研究，你怎样判断自己研究的方向和做出的各种选择是否正确?

Andrew Ng：诚然，神经科学尚未揭开人脑的运作机制，是对这项研究的一种制约。但如今我们尝试的算法，大多只是粗略地基于神经科学研究的统计阐释，这些研究启迪 我们的灵感，鼓励我们尝试新算法。但现实中，我们更主要地依据算法真实的运行效果进行评判，假如一味追求模拟神经的运作方式，不一定能带来最优的结果。有 时我们偏重神经科学原理，例如

某些模拟大脑局部的算法；但更多时候，性能是准绳。若按比例划分，前者大约只占2%，后者则占据98%。

因为 我们不知道何种算法最优，所以才不断尝试，衡量是否取得进步的方式之一（并非唯一方式），是观察新算法能否在应用中表现得更好，例如Web和图像搜索结果 是否更准确，或者语音识别的正确率更高。假如回望五年，你就会发现，那时我们曾认为颇有前景的算法，如今已然被抛弃。这些年，我们有规律地，甚至偶然地发 现一些新算法，推动着这个领域持续前进。

《程序员》：关于Deep Learning的原理，已有许多人知晓。为了做出一流研究和应用，对于研究者来说，决胜的关键因素是什么？为何如今只有少数几人，成为这个领域的顶尖科学家？

Andrew Ng：关于决定因素，我认为有三点最为关键。

首先是数据，对于解决某些领域的问题，获取数据并非轻而易举；其次是计算基础架构工具，包括计算机硬件和软件；最后是这个领域的工程师培养，无论在斯坦福还 是百度，我都对如何快速训练工程师从事Deep Learning研究，成为这个领域的专家思索了很长时间。幸运的是，我从Coursera和大学的教学经历中获益良多。创新往往来自多个观念的整合，源 于一整支研究团队，而非单独一个人。

从事Deep Learning研究的一个不利因素在于，这还是一个技术快速发展的年轻领域，许多知识并非依靠阅读论文便能获得。那些关键知识，往往只存在于顶尖科学家 的头脑中，这些专家彼此相识，信息相互共享，却不为外人所知。另外一些时候，这群顶尖科学家也不能确定自己的灵感源于何处，如何向其他人解释。但我相信， 越来越多的知识会传递给普通开发者；在百度，我也正努力寻找方法，将自己的灵感和直觉高效地传授给其他研究者。尽管我们已有这方面的教程，但需要改进之处 仍有很多。

此外，许多顶尖实验室的迭代速度都非常快，而Deep Learning算法复杂，计算代价很高，这些实验室都拥有优秀的工具与之配合，从快速迭代中学习进步。

《程序员》：十年前，Jeff Hawkins在《On Intelligence》中已经向普通人描述了机器与智能之间的关系，这些描述与我们现在看到的Deep Learning似乎非常相似。在这最近的十年中，我们新学到了什么?

Andrew Ng：包括我在内，Jeff Hawkins的作品启发了许多AI研究者，多年以前，我个人还曾是Hawkins这家公司的技术顾问之一。但在现实中，每个人的实现细节和算法迥异，与 这本书其实颇有不同。例如在书中，Hawkins极为强调与时间相关的临时数据的重要性，而在Deep Learning中，虽然用到了临时数据，但远没有那重要，另外网络的架构也大不相同。在最近十年中，我们认识到了可扩展的重要性，另外我们还找到了进行 非监督式学习更好的方式。

关于工作选择

《程序员》：为什么选择百度开展你的工作，它的哪些特点，是你觉得其他公司所不具备的?

Andrew Ng：我喜欢在任务高度驱动的环境下工作，通常我是这些任务的发起者。我为能更好地发展AI，令互联网上的每个人都能从中受益而兴奋。

几个月前，我仔细评估了几个选项后，决定加入百度。一方面在于王劲团队打造了非常优秀的基础设施，同时百度还拥有庞大的数据。另一方面，我为百度的灵活快速 所吸引，当我的朋友余凯和徐伟决定搭建GPU集群，马上就得到了实现，此外没有任何一家公司推出Deep Learning产品的速度快过这里，而且还是应用在互联网广告这种核心业务上。对于Deep Learning这样未知因素很多的技术来说，灵活性至关重要。我还发现北京的互联网公司讨论的往往是日活跃用户，而在硅谷则是月活跃用户，或许这也可以 作为灵活性的一个注解。

还有一点我很少谈起，却非常重要——因为这里的人。与他们相处，我感到非常愉快。当我开始在百度工作后，妻子Carol曾对我说，她从未见过我如此努力，却又如此开心。

《程序员》：你在百度的研究产品和成果能为外界带来什么（例如是未来否有可能将你的成果共享给其他人，推动整个领域的发展）?

Andrew Ng：我希望能将成果与外界分享，也许不是所有内容都适合，但希望通过某种形式，分享我们的研究。不过我加入的时间尚短，接下来我希望能有更多成果可以公布。

原文链接：http://www.csdn.net/article/2014-08-01/2821007

iOS 通知中心扩展制作入门

作者：OneV’s Den

总览

扩展 (Extension) 是 iOS 8 和 OSX 10.10 加入的一个非常大的功能点，开发者可以通过系统提供给我们的扩展接入点 (Extension point) 来为系统特定的服务提供某些附加的功能。对于 iOS 来说，可以使用的扩展接入点有以下几个：

- Today 扩展 - 在下拉的通知中心的 "今天" 的面板中添加一个 widget
- 分享扩展 - 点击分享按钮后将网站或者照片通过应用分享
- 动作扩展 - 点击 Action 按钮后通过判断上下文来将内容发送到应用
- 照片编辑扩展 - 在系统的照片应用中提供照片编辑的能力
- 文档提供扩展 - 提供和管理文件内容
- 自定义键盘 - 提供一个可以用在所有应用的替代系统键盘的自定义键盘或输入法

系统为我们提供的接入点虽然还比较有限，但是不少已经是在开发者和 iOS 的用户中呼声很高的了。而通过利用这些接入点提供相应的功能，也可以极大地丰富系统的功能和可用性。本文将先不失一般性地介绍一下各种扩展的共通特性，然 后再以一个实际的例子着重介绍下通知中心的 Today 扩展的开发方法，以期为 iOS 8 的扩展的学习提供一个平滑的入口。

Apple 指出，iOS 8 中开发者的中心并不应该发生改变，依然应该是围绕 app。在 app 中提供优秀交互和有用的功能，现在是，将来也会是 iOS 应用开发的核心任务。而扩展在 iOS 中是不能以单独的形式存在的，也就是说我们不能直接在 AppStore 提供一个扩展的下载，扩展一定是随着一个应

用一起打包提供的。用户在安装了带有扩展的应用后，将可以在通知中心的今日界面中，或者是系统的设置中来选择开 启还是关闭你的扩展。而对于开发者来说，提供扩展的方式是在 app 的项目中加入相应的扩展的 target。因为扩展一般来说是展现在系统级别的 UI 或者是其他应用中的，Apple 特别指出，扩展应该保持轻巧迅速，并且专注功能单一，在不打扰或者中断用户使用当前应用的前提下完成自己的功能点。因为用户是可以自己选择禁用扩展的，所 以如果你的扩展表现欠佳的话，很可能会遭到用户弃用，甚至导致他们将你的 app 也一并卸载。

扩展的生命周期

扩展的生命周期和包含该扩展的你的容器 app (container app) 本身的生命周期是独立的，准确地说。它们是两个独立的进程，默认情况下互相不应该知道对方的存在。扩展需要对宿主 app (host app，即调用该扩展的 app) 的请求做出相应，当然，通过进行配置和一些手段，我们可以在扩展中访问和共享一些容器 app 的资源，这个我们稍后再说。

因为扩展其实是依赖于调用其的宿主 app 的，因此其生命周期也是由用户在宿主 app 中的行为所决定的。一般来说，用户在宿主 app 中触发了该扩展后，扩展的生命周期就开始了：比如在分享选项中选择了你的扩展，或者向通知中心中添加了你的 widget 等等。而所有的扩展都是由 ViewController 进行定义的，在用户决定使用某个扩展时，其对应的 ViewController 就会被加载，因此你可以像在编写传统 app 的 ViewController 那样获取到诸如 viewDidLoad 这样的方法，并进行界面构建及做相应的逻辑。扩展应该保持功能的单一专注，并且迅速处理任务，在执行完成必要的任务，或者是在后台预约完成任务后，一般需要尽快通过回调将控制权交回给宿主 app，至此生命周期结束。

按照 Apple 的说法，扩展可以使用的内存是远远低于 app 可以使用的内存的。在内存吃紧的时候，系统更倾向于优先搞掉扩展，而不会是把宿主 app 杀死。因此在开发扩展的时候，也一定需要注意内存占用的限制。另一点是比如像通知中心扩展，你的扩展可能会和其他开发人员的扩展共存，这样如果扩展阻塞了 主线程的话，就会引起整个通知中心失去响应。这种情况下你的扩展和应用也就基本和用户说再见了..

扩展和容器应用的交互

扩展和容器应用本身并不共享一个进程，但是作为扩展，其实是主体应用功能的延伸，肯定不可避免地需要使用到应用本身的逻辑甚至界面。在这种情况下， 我们可以使用 iOS 8 新引入的自制 framework 的方式来组织需要重用的代码，这样在链接 framework 后 app 和扩展就都能使用相同的代码了。

另一个常见需求就是数据共享，即扩展和应用互相希望访问对方的数据。这可以通过开启 App Groups 和进行相应的配置来开启在两个进程间的数据共享。这包括了使用 NSUserDefaults 进行小数据的共享，或者使用 NSFileCoordinator 和 NSFilePresenter 甚至是 CoreData 和 SQLite 来进行更大的文件或者是更复杂的数据交互。

另外，一直以来的自定义的 url scheme 也是从扩展向应用反馈数据和交互的渠道之一。

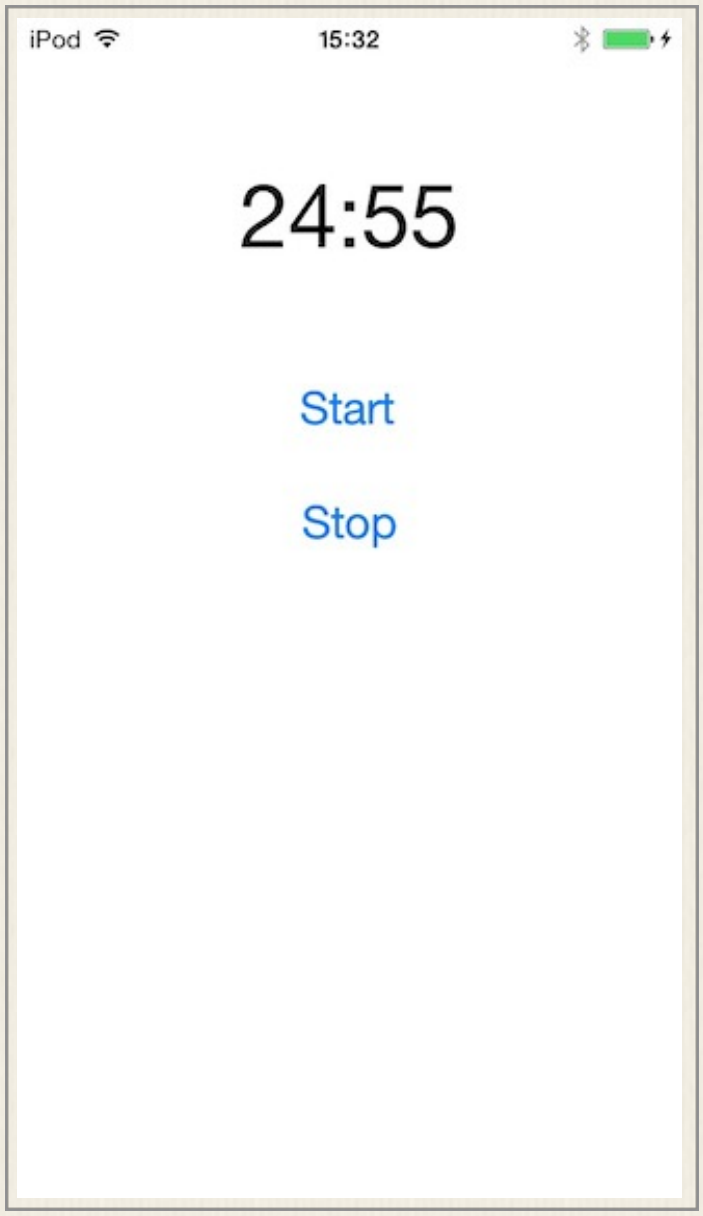
这些常见的手段和策略在接下来的 demo 中都会用到。一张图片能顶千言万语，而一个 demo 能顶千张图片。那么，我们开始吧。

Timer Demo

Demo 做的应用是一个简单的计时器，即点击开始按钮后开始倒数计时，每秒根据剩余的时间来更新界面上的一个表示时间的 Label，然后在计时到 0 秒时弹出一个 alert，来告诉用户时间到，当然用户也可以使用 Stop 按钮来提前打断计时。其实这个 Demo 就是我的很早之前做的一个番茄工作法的 app 的原型。

为了大家方便跟随这个 demo，我把初始的时候的代码放到 GitHub 的 start-project 这个 tag 上了。语言当然是要用 Swift，界面因为不是 demo 的重点，所以就非常简单能表明意思就好了。但是虽然简单，却也是利用了上一篇文章中所提到的 Size Classes 来完成的不同屏幕的布局，所以至少可以说在思想上是完备的 iOS 8 兼容了 =_=..

初始工程运行起来的界面大概是这样的：



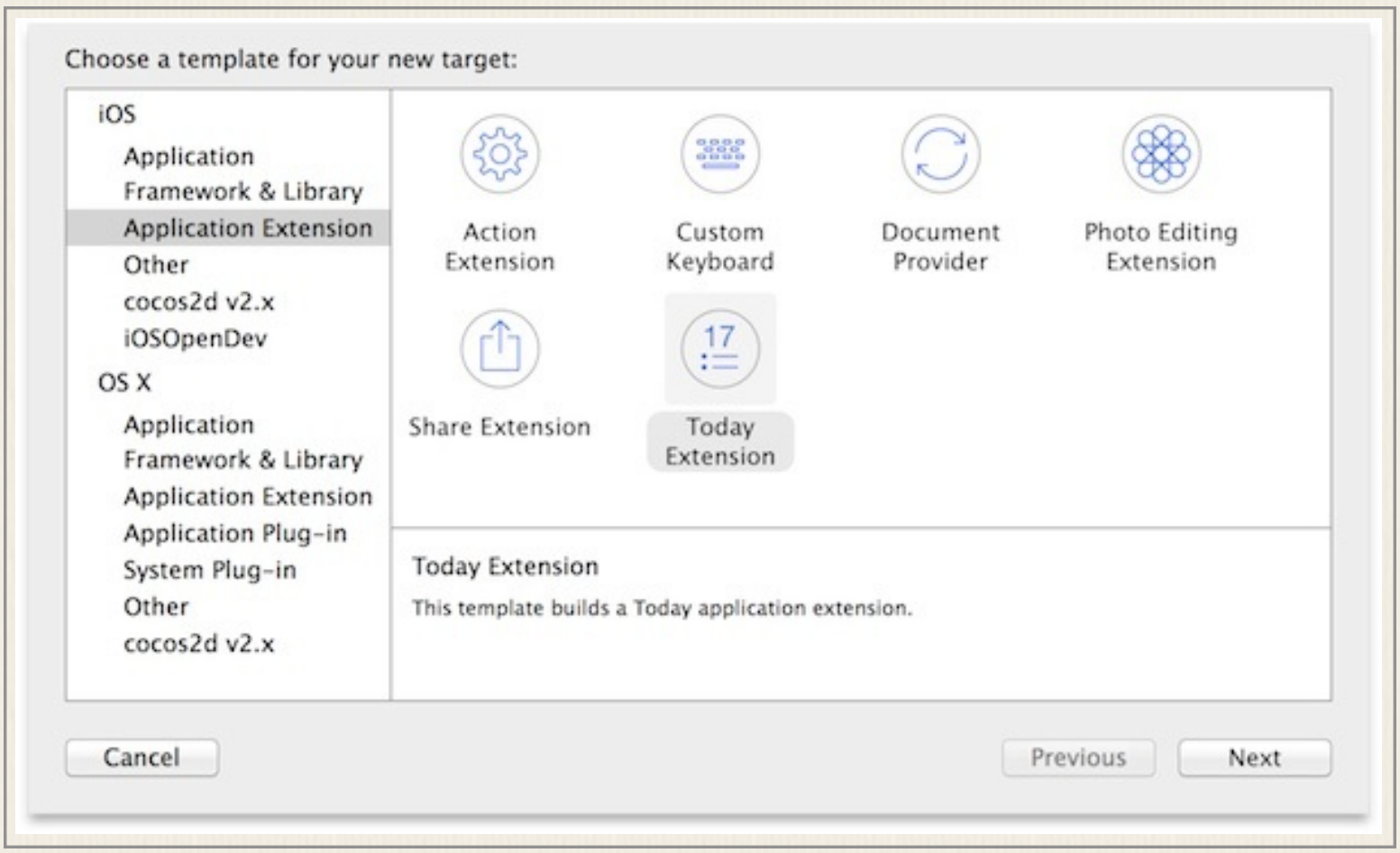
简单说整个项目只有一个 ViewController，点击开始按钮时我们通过设定希望的计时时间来创建一个 Timer 实例，然后调用它的 start 方法。这个方法接收两个参数，分别是每次剩余时间更新，以及计时结束（不论是计时时间到的完成还是计时被用户打断）时的回调方法。另外这个方法返回一个 tuple，用来表示是否开始成功以及可能的错误。

剩余时间更新的回调中刷新界面 UI，计时结束的回调里回收了 Timer 实例，并且显示了一个 UIAlertController。用户通过点击 Stop 按钮可以直接调用 stop 方法来打断计时。直接简单，没什么其他的 trick。

我们现在计划为这个 app 做一个 Today 扩展，来在通知中心中显示并更新当前的剩余时间，并且在计时完成后显示一个按钮，点击后可以回到 app 本体，并弹出一个完成的提示。

添加扩展 Target

第一步当然是为我们的 app 添加扩展。正如在总览中所提到的，扩展是项目中的一个单独的 target。在 Xcode 6 中， Apple 为我们准备了对应各类不同扩展点的 target 模板，这使得向 app 中添加扩展非常容易。对于我们现在想做的 Today 扩展，只需点选菜单的 File > New > Target...，然后选择 iOS 中的 Application Extension 的 Today Extension 就行了。



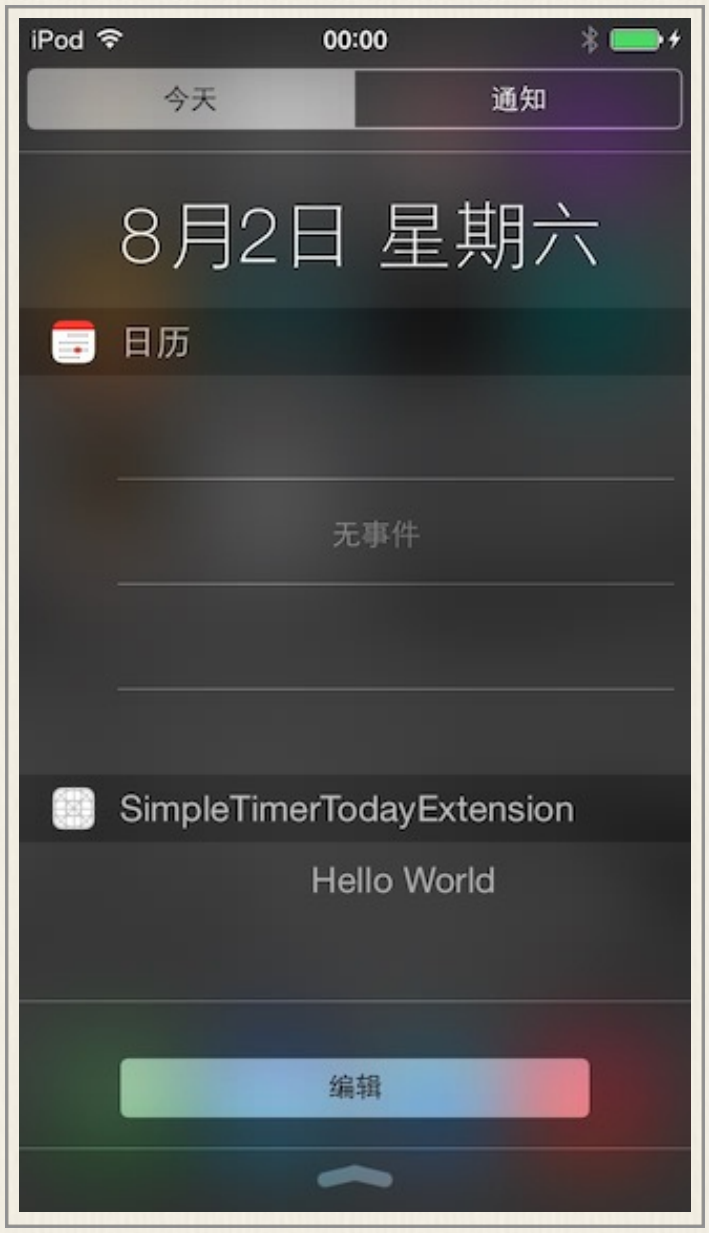
在弹出的菜单中将新的 target 命名为 SimpleTimerTodayExtension，并且让 Xcode 自动生成新的 Scheme ，以方便测试使用。我们的工程中现在会多出一个和新建的 target 同名的文件夹，里面主要包含了一个 .swift 的 ViewController 程序文件，一个叫做 MainInterface 的 storyboard 文件和 Info.plist。其中在 plist 里 的 NSExtension 中定义了这个 扩展的类型和入口，而配套的 ViewController 和 StoryBoard 就是我们的扩展的具体内容和实现了。

我们的主题程序在编译链接后会生成一个后缀为 .app 的包，里面包含主程序的二进制文件和各种资源。而扩展 target 将单独生成一个后缀名为 .appex 的文件包。这个文件包将随着主体程序被安装，并由用户选择激活或者

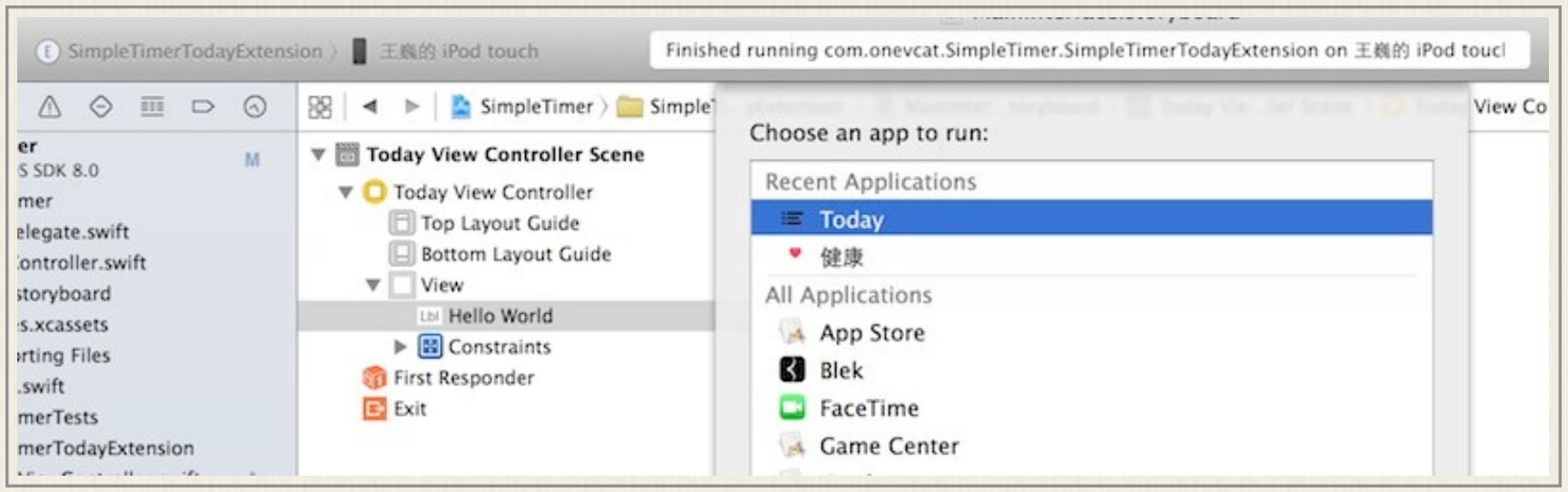
添加（对于 Today widget 的话在通知中心 Today 视图中的编辑删增，对于其他的扩展的话，使用系统的设置进行管理）。我们可以看到，现在项目的 Product 中已经新增了一个扩展了。

如果你有心已经打开了 MainInterface 文件的话，可以注意到 Apple 已经为我们准备了一个默认的 Hello World 的 label 了。我们这时候只要运行主程序，扩展就会一并安装了。将 Scheme 设为 Simple Timer 的主程序，Cmd + R，然后点击 Home 键将 app 切到后台，拉下通知中心。这时候你应该能在 Toady 视图中找到叫做 SimpleTimerTodayExtenstion 的项目，显示了一个 Hello World 的标签。如果没有的话，可以点击下面的编辑按钮看看是不是没有启用，如果在编辑菜单中也没有的话，恭喜你遇到了和 Session 视频里的演讲者同样的 bug，你可能需要删除应用，清理工程，然后再安装试试看。一般来说卸载再安装可以解决现在的 beta 版大部分的无法加载的问题，如果还是遇到问题的话，你还可以尝试重启设备（按照以往几年的 SDK 的情况来看，beta 版里这很正常，正式版中应该就没什么问题了）。

如果一切正常的话，你能看到的通知中心应该类似这样：



这种方式运行的扩展我们无法对其进行调试，因为我们的调试器并没有 attach 到这个扩展的 target 上。有两种方法让我们调试扩展，一种是将 Scheme 设为之前 Xcode 为我们生成的 SimpleTimerTodayExtenstion， 然后运行时选择从 Today 视图进行运行，如图；另一种是在扩展运行时使用菜单中的 Debug > Attach to Process > By Process Identifier (PID) or name，然后输入你的扩展的名字（在我们的 demo 中是 com.onevcat.SimpleTimer.SimpleTimerTodayExtension）来把调试器挂载到进程上去。

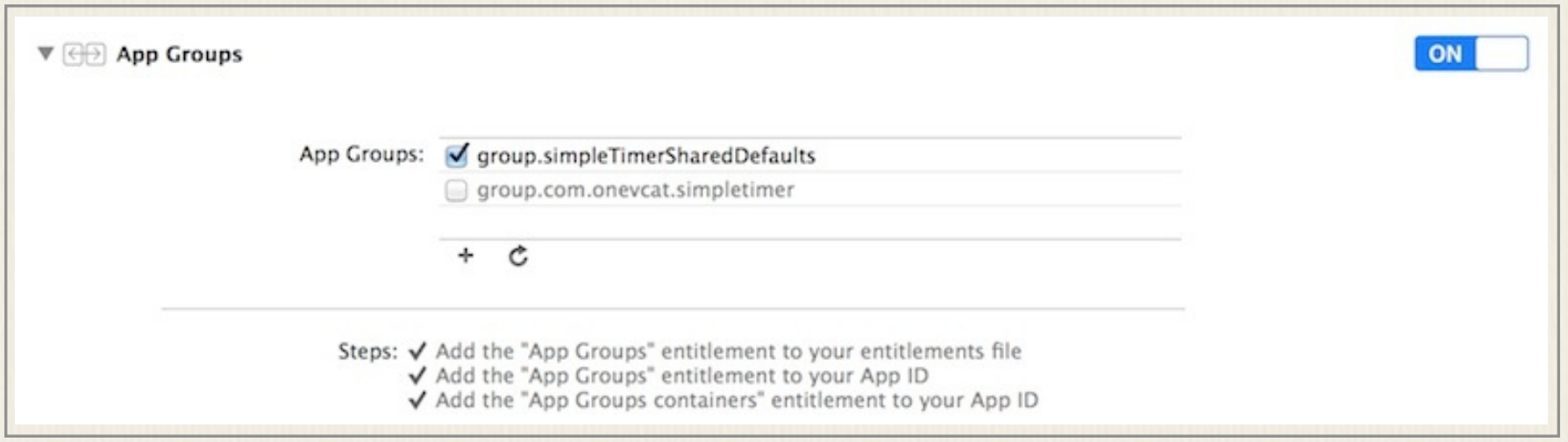


在应用和扩展间共享数据 - App Groups

扩展既然是个 ViewController，那各种连接 IBOutlet，使用 viewDidLoad 之类的生命周期方法来设置 UI 什么的自然不在话下。我们现在的第一个难点就是，如何获取应用主体在退出时计时器的剩余时间。只要知道了还剩多久以及何时退出，我们就能在通知中心中显示出计时器正确的剩余时间了。

对 iOS 开发者来说，沙盒限制了我们在设备上随意读取和写入。但是对于应用和其对应的扩展来说，Apple 在 iOS 8 中为我们提供了一种可能性，那就是 App Groups。App Groups 为同一个 vender 的应用或者扩展定义了一组域，在这个域中同一个 group 可以共享一些资源。对于我们的例子来说，我们只需要使用同一个 group 下的 NSUserDefaults 就能在主体应用不活跃时向其中存储数据，然后在扩展初始化时从同一处进行读取就行了。

首先我们需要开启 App Groups。得益于 Xcode 5 开始引入的 Capabilities，这变得非常简单（至少不再需要去 developer portal 了）。选择主 target SimpleTimer，打开它的 Capabilities 选项卡，找到 App Groups 并打开开关，然后添加一个你能记得的 group 名字，比如 group.simpleTimerSharedDefaults。接下来你还需要为 SimpleTimerTodayExtension 这个 target 进行同样的配置，只不过不再需要新建 group，而是勾选刚才创建的 group 就行。



然后让我们开始写代码吧！首先是在主体程序的 *ViewController.swift* 中添加一个程序失去前台的监听，在 *viewDidLoad* 中加入：

```
NSNotificationCenter.defaultCenter()
    .addObserver(self, selector: "applicationWillResignActive",name:
UIApplicationWillResignActiveNotification, object: nil)
```

然后是所调用的 *applicationWillResignActive* 方法：

```
@objc private func applicationWillResignActive() {
    if timer == nil {
        clearDefaults()
    } else {
        if timer.running {
            saveDefaults()
        } else {
            clearDefaults()
        }
```

```
        }
    }

    private func saveDefaults() {
        let userDefault = NSUserDefaults(suiteName:
"group.simpleTimerSharedDefaults")
        userDefault.setInteger(Int(timer.leftTime), forKey:
"com.onevcat.simpleTimer.lefttime")
        userDefault.setInteger(Int(NSDate().timeIntervalSince1970), forKey:
"com.onevcat.simpleTimer.quitdate")

        userDefault.synchronize()
    }
```

```
    private func clearDefaults() {
        let userDefault = NSUserDefaults(suiteName:
"group.simpleTimerSharedDefaults")

userDefault.removeObjectForKey("com.onevcat.simpleTimer.lefttime")

userDefault.removeObjectForKey("com.onevcat.simpleTimer.quitdate")

        userDefault.synchronize()
    }
```

这样，在应用切到后台时，如果正在计时，我们就将当前的剩余时间和退出时的日期存到了 NSUserDefaults 中。这里注意，可能一般我们在使用 NSUserDefaults 时更多地是使用 standardUserDefaults，但是这里我们需要这两个数据能够被扩展访问到的话，我们必须使用在 App Groups 中定义的名字来使用 NSUserDefaults。

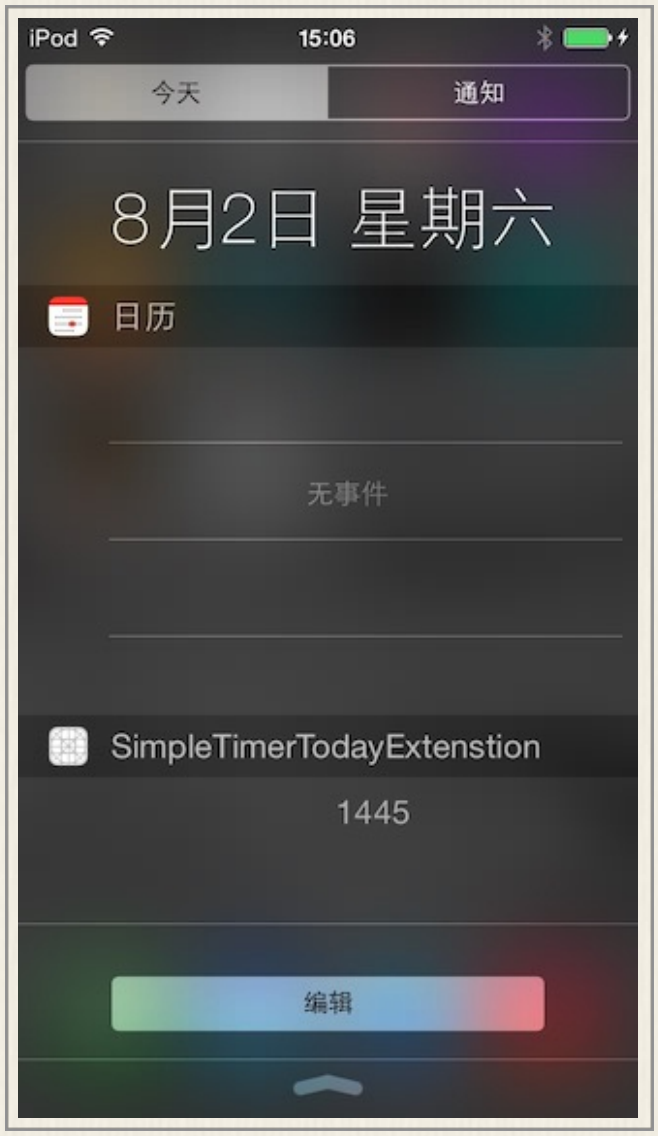
接下来，我们可以到扩展的 TodayViewController.swift 中去获取这些数据了。在扩展 ViewController 的 viewDidLoad 中，添加以下代码：

```
let userDefaults = NSUserDefaults(suiteName: "group.simpleTimerSharedDefaults")

let leftTimeWhenQuit = userDefaults.integerForKey("com.onevcat.simpleTimer.lefttime")

let quitDate = userDefaults.integerForKey("com.onevcat.simpleTimer.quitdate")

let passedTimeFromQuit = NSDate().timeIntervalSinceDate(NSDate(timeIntervalSince1970: NSTimeInterval(quitDate)))

let leftTime = leftTimeWhenQuit - Int(passedTimeFromQuit)

lblTImer.text = "\(leftTime)"
```

当然别忘了把 StoryBoard 的那个 label 拖出来：

```
@IBOutlet weak var lblTImer: UILabel!
```

再次运行程序，并开始一个计时，然后按 Home 键切到后台，拉出通知中心，perfect，我们的扩展能够和主程序进行数据交互了：



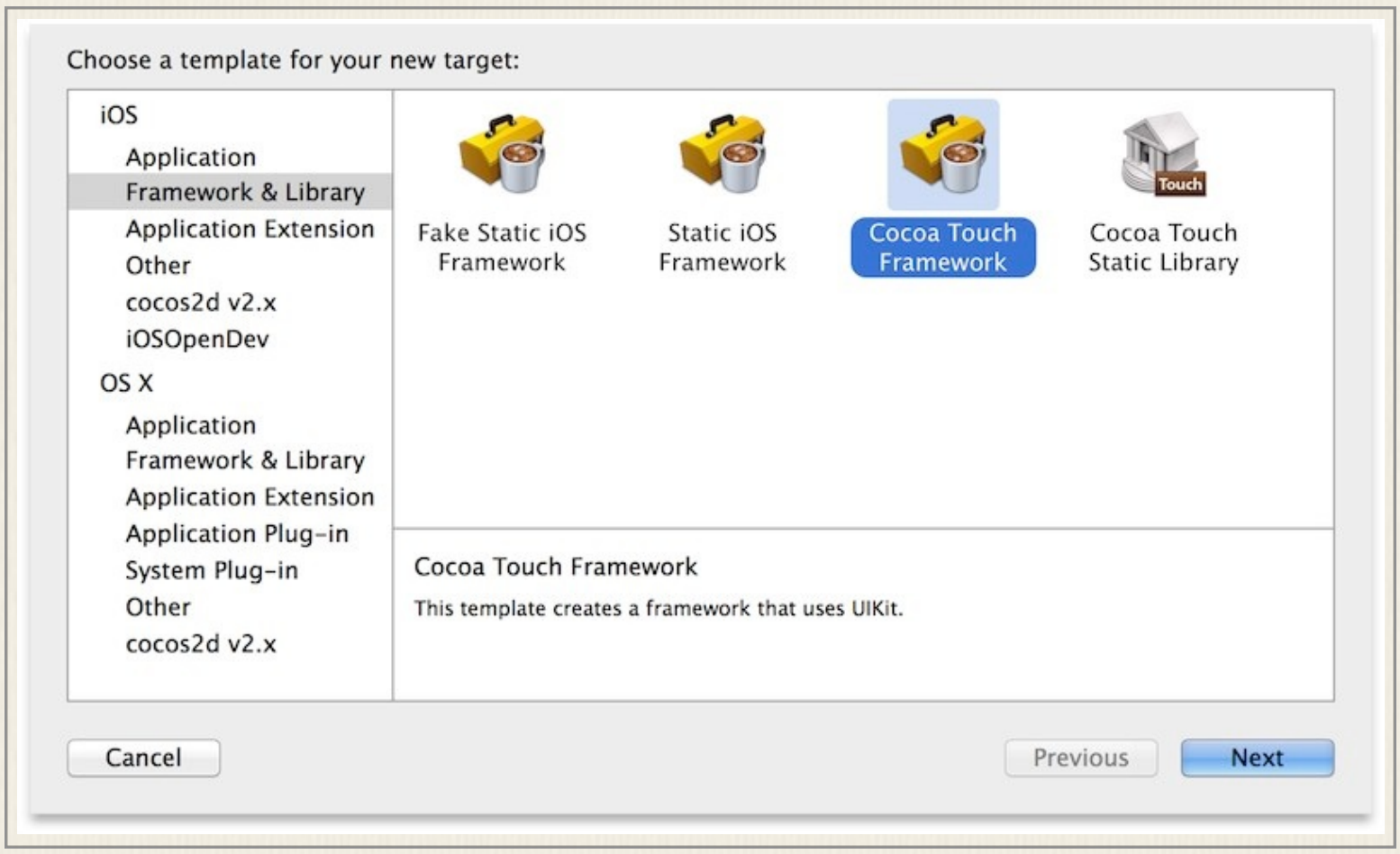
在应用和扩展间共享代码 - Framework

接下来的任务是在 Today 界面中进行计时，来刷新我们的界面。这部分代码其实我们已经写过（当然..确切来说是我写的，你可能只是看过），没错，就是应用中的 Timer.swift 文件。我们只需要在扩展的 ViewController 中用剩余时间创建一个 Timer 的实例，然后在更新的 callback 里设置 label 就好了嘛。但是问题是，这部分代码是在应用中的，我们要如何在扩展中也能使用它呢?

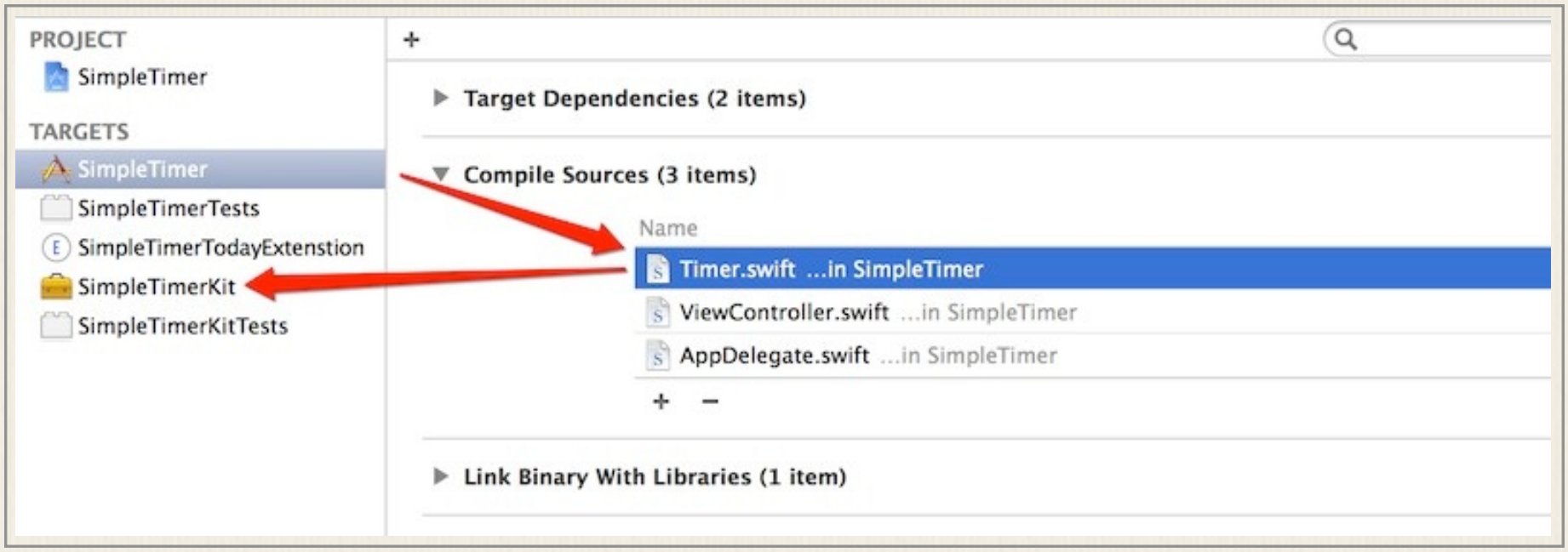
一个最直接也是最简单的想法自然是把 Timer.swift 加入到扩展 target 的编译文件中去，这样在扩展中自然也就可以使用了。但是 iOS 8 开始 Apple 为我们提供了一个更好的选择，那就是做成 Framework。单个文件可能不会觉得有什么差别，但是随着需要共用的文件数量和种类的增加，将单个文件逐一添加到不同 target 这种管理方法很快就会将事情弄成一团乱麻。你需要考虑每一个新加或者删除的文件影响的范围，以及它们分别需要适用何处，这简直就是人间地狱。提供一个统一 漂亮的 framework 会是更多人希

望的选择（其实也差不多成为事实标准了）。使用 framework 进行模块化的另一个好处是可以得益于良好的访问控制，以保证你不会接触到不应该使用的东西，然后，Swift 的 namespace 是基于模块的，因此你也不再需要担心命名冲突等等一摊子 objc 时代的烦心事儿。

现在让我们把 Timer.swift 放到 framework 里吧。首先我们新建一个 framework 的 target。File > New > Target... 中选择 Framework & Library，选中 Cocoa Touch Framework （配图中的另外几个选项可能在你的 Xcode 中是没有的，请无视它们，这是历史遗留问题），然后确定。按照 Apple 对 framework 的命名规范，也许 SimpleTimerKit 会是一个不错的名字。

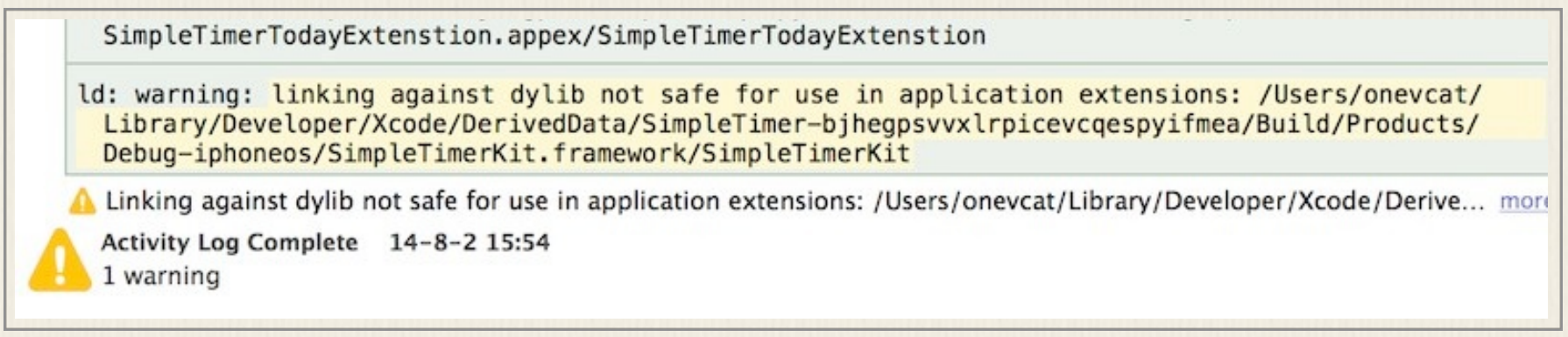


接下来，我们将 Timer.swift 从应用中移动到 framework 中。很简单，首先将其从应用的 target 中移除，然后加入到新建的 SimpleTimerKit 的 Compile Sources 中。

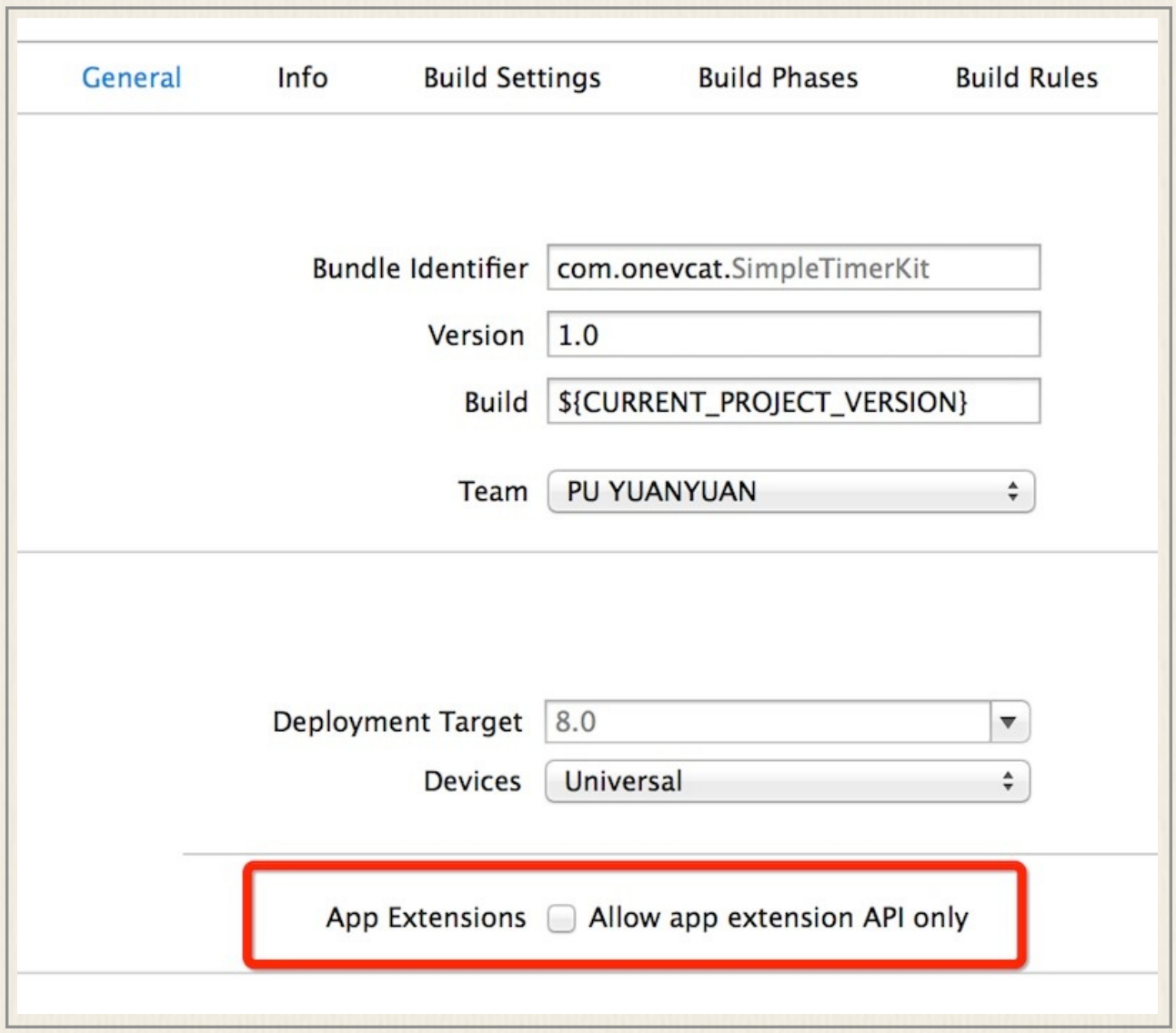


确认在应用中 link 了新的 framwork，并且在 ViewController.swift 中加上 import SimpleTimerKit 后试着编译看看...好多错误，基本都是 ViewController 中说找不到 Timer 之类的。这是因为原来的实现是在同一个 module 中的，默认的 internal 的访问层级就可以让 ViewController 访问到关于 Timer 和相应方法的信息。但是现在它们处于不同的 module 中，所以我们需要对 Timer.swift 的访问权限进行一些修改，在需要外部访问的地方加上 public 关键字。关于 Swift 中的访问控制，可以参考 Apple 关于 Swift 的这篇官方博客，简单说就是 private 只允许本文件访问，不写的话默认是 internal，允许统一 module 访问，而要提供给别的 module 使用的话，需要声明为 public。修改后的 Timer.swift 文件大概是这个样子的。

修改合适的访问权限后，接下来我们就可以将这个 framework 链接到扩展的 target 了。链接以后编译什么的可以通过，但是会多一个警告：



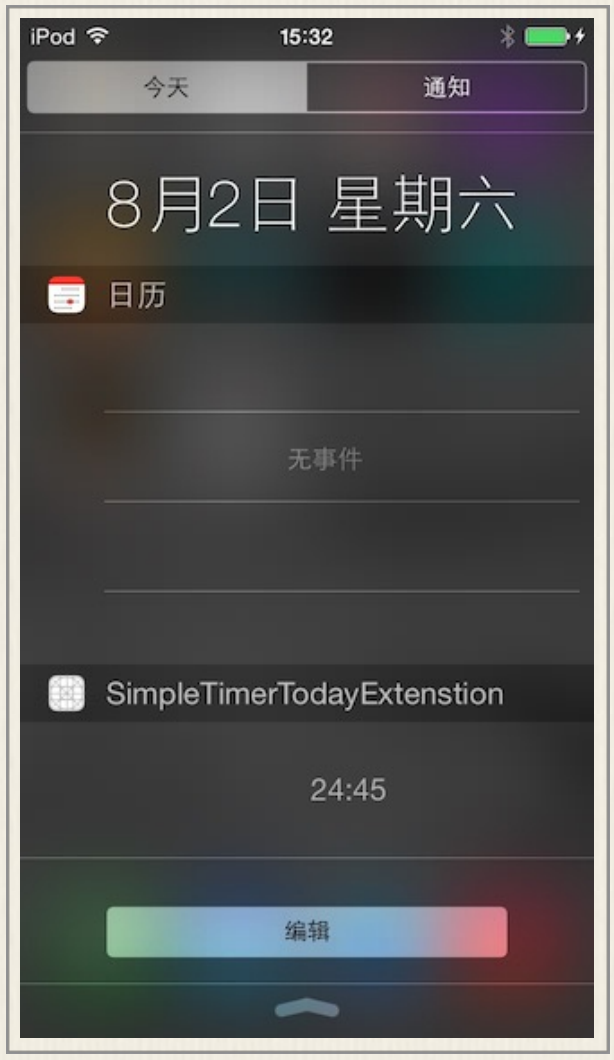
这是因为作为插件，需要遵守更严格的沙盒限制，所以有一些 API 是不能使用的。为了避免这个警告，我们需要在 framework 的 target 中声明在我们使用扩展可用的 API。具体在 SimpleTimerKit 的 target 的 General 选项卡中，将 Deployment Info 中的 Allow app extension API only 勾选上就可以了。关于在扩展里不能使用的 API，都已经被 Apple 标上了 NS_EXTENSION_UNAVAILABLE，在这里有一份简单的列表可供参考，基本来说都是 runtime 的东西以及一些会让用户迷惑或很危险的操作（当然这个标记的方法很可能会不断变动，最终一切请以 Apple 的文档和实际代码为准）。



接下来，在扩展的 ViewController 中也链接 SimpleTimerKit 并加入 import SimpleTimerKit，我们就可以在扩展中使用 Timer 了。将刚才的直接设置 label 的代码去掉，换成下面的：

```
override func viewDidLoad() {
    //...

    if (leftTime > 0) {
        timer = Timer(timeInteral: NSTimeInterval(leftTime))
        timer.start(updateTick: {
                [weak self] leftTick in self!.updateLabel()
            }, stopHandler: nil)
    } else {
        //Do nothing now
    }
}

    private func updateLabel() {
        lblTimer.text = timer.leftTimeString
    }
```

我们在扩展里也像在 app 内一样，创建 Timer，给定回调，坐等界面刷新。运行看看，先进入应用，开始一个计时。然后退出，打开通知中心。通知中心中现在也开始计时了，而且确实是从剩余的时间开始的，一切都很完美：



通过扩展启动主体应用

最后一个任务是，我们想要在通知中心计时完毕后，在扩展上呈现一个 "完成啦" 的按钮，并通过点击这个按钮能回到应用，并在应用内弹出结束的 alert。

这其实最关键的在于我们要如何启动主体容器应用，以及向其传递数据。可能很多同学会想到 URL Scheme，没错通过 URL Scheme 我们确实可以启动特定应用并携带数据。但是一个问题是为了通过 URL 启动应用，我们一般需要调用 UIApplication 的 openURL 方法。如果细心的刚才看了 NS_EXTENSION_UNAVAILABLE 的同学可能会发现这个方法是被禁用的（这也是很 make sense 的一件事情，因为说白了扩展通过 sharedApplication 拿到的其实是宿主应用，宿主应用表示凭什么要让你拿到啊！）。为了完成同样的操作，Apple 为扩展提供了一个 NSExtensionContext 类来与宿主应用进行交互。用户在宿主应用中启动扩展后，宿主应用提供一个上下文给扩展，里面最主要的是包含了 inputItems 这样的待处理的数据。当然对

我们现在的需求来说，我们只要用到它的 openURL(URL:,completionHandler:) 方法就好了。

另外，我们可能还需要调整一下扩展 widget 的尺寸，以让我们有更多的空间显示按钮，这可以通过设定 preferredContentSize 来做到。在 TodayViewController.swift 中加入以下方法：

```
private func showOpenAppButton() {
    lblTimer.text = "Finished"
    preferredContentSize = CGSizeMake(0, 100)

    let button = UIButton(frame: CGRectMake(0, 50, 50, 63))
    button.setTitle("Open", forState: UIControlState.Normal)
    button.addTarget(self, action: "buttonPressed:", forControlEvents: UIControlEvents.TouchUpInside)
    view.addSubview(button)
}
```

在设定 preferredContentSize 时，指定的宽度都是无效的，系统会自动将其处理为整屏的宽度，所以扔个 0 进去就好了。在这里添加按钮时我偷了个懒，本来应该使用Auto Layout 和添加约束的，但是这并不是我们这个 demo 的重点。另一方面，为了代码清晰明了，就直接上坐标了。

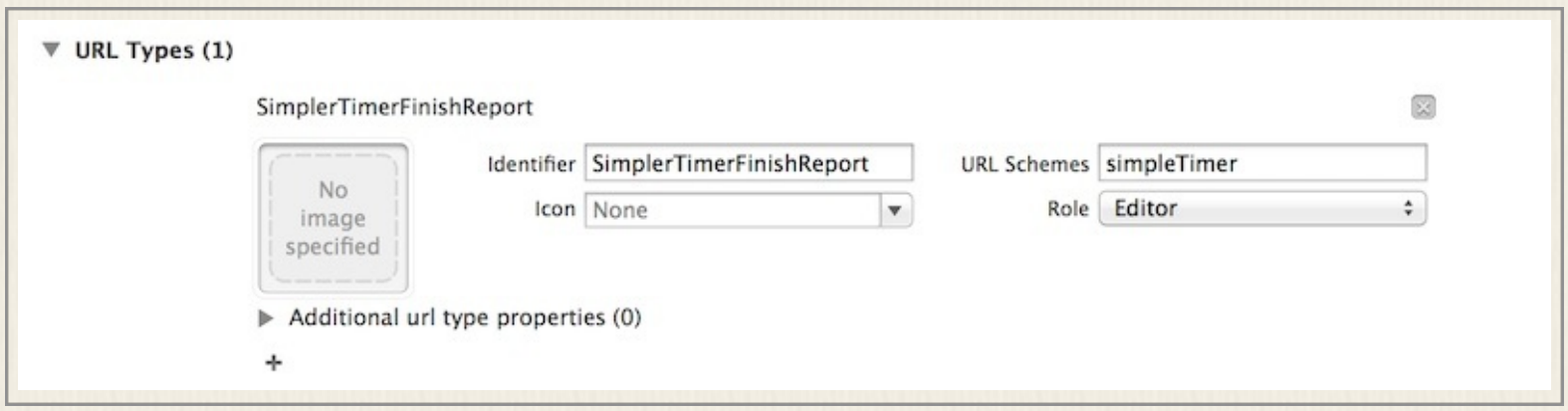
然后添加这个按钮的 action：

```
@objc private func buttonPressed(sender: AnyObject!) {
    extensionContext.openURL(NSURL(string: "simpleTimer://finished"), completionHandler: nil)
}
```

我们将传递的 URL 的 scheme 是 simpleTimer，以 host 的 finished 作为参数，就可以通知主体应用计时完成了。然后我们需要在计时完成时调用 showOpenAppButton 来显示按钮，更新 viewDidLoad 中的内容：

```
override func viewDidLoad() {
    //...
    if (leftTime > 0) {
        timer = Timer(timeInteral: NSTimeInterval(leftTime))
        timer.start(updateTick: {
            [weak self] leftTick in self!.updateLabel()
            }, stopHandler: {
                [weak self] finished in self!.showOpenAppButton()
            })
    } else {
        showOpenAppButton()
    }
}
```

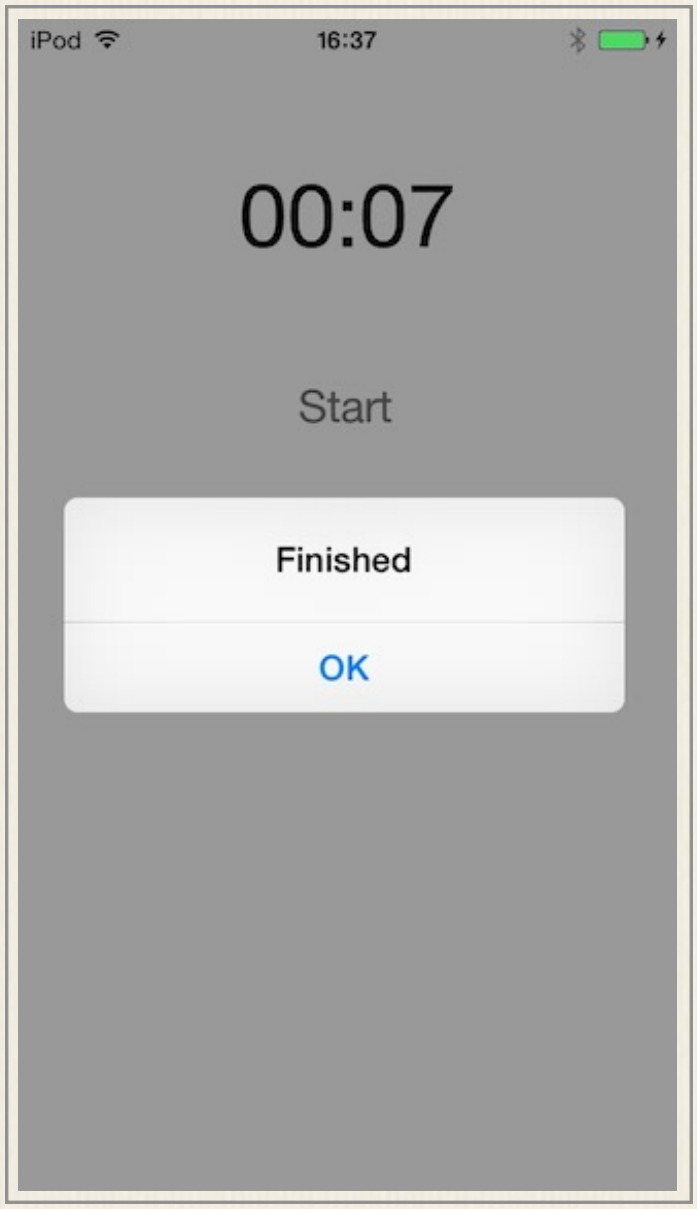
最后一步是在主体应用的 target 里设置合适的 URL Scheme：



然后在 AppDelegate.swift 中捕获这个打开事件，并检测计时是否完成，然后做出相应：

```
func application(application: UIApplication!, openURL url: NSURL!, sourceApplication: String!, annotation: AnyObject!) -> Bool {
    if url.scheme == "simpleTimer" {
        if url.host == "finished" {
            NSNotificationCenter.defaultCenter()
                .postNotificationName(taskDidFinishedInWidgetNotification, object: nil)
        }
        return true
    }
    return false
}
```

在这个例子里，我们发了个通知。而在 ViewController 中我们可以一开始就监听这个通知，然后收到后停止计时并弹出提示就行了。当然我们可能需要一些小的重构，比如添加是手动打断还是计时完成的判断以弹出不一样的对话框等等，这些都很简单再次就不赘述了。



至此，我们就完成了一个很基本的通知中心扩展，完整的项目可以在 GitHub repo 的 master 上找到。这个计时器现在在应用中只在前台或者通知中心显示时工作，如果你退出应用后再打开应用，其实这段时间内是没有计时的。因此这个项目之后可能的改进就是在返回应用的时候添加一下计时的判定，来更新计时器的剩余时间，或者是已经完成了的话就直接结束计时。

其他

其实在 Xcode 为我们生成的模板文件中，还有这么一段代码也很重要：

```
func widgetPerformUpdateWithCompletionHandler(completionHandler: ((NCUpdateResult) -> Void)!) {

    // Perform any setup necessary in order to update the view.
```

```
    // If an error is encoutered, use NCUpdateResult.Failed
    // If there's no update required, use NCUpdateResult.NoData
    // If there's an update, use NCUpdateResult.NewData

    completionHandler(NCUpdateResult.NewData)
}
```

对于通知中心扩展，即使你的扩展现在不可见 (也就是用户没有拉开通知中心)，系统也会时不时地调用实现了 NCWidgetProviding 的扩展的这个方法，来要求扩展刷新界面。这个机制和 iOS 7 引入的后台机制是很相似的。在这个方法中我们一般可以做一些像 API 请求之类的事情，在获取到了数据并更新了界面，或者是失败后都使用提供的 completionHandler 来向系统进行报告。

值得注意的一点是 Xcode (至少现在的 beta 4) 所提供的模板文件的 ViewController 里虽然有这个方法，但是它默认并没有 conform 这个接口，所以要用的话，我们还需要在类声明时加上 NCWidgetProviding。

总结

这个 Demo 主要涉及了通知中心的 Toady widget 的添加和一般交互。其实扩展是一个相当大块的内容，对于其他像是分享或者是 Action 的扩展，其使用方式又会有所不同。但是核心的概念，生命周期以及与本体应用交互的方法都是相似的。Xcode 在我们创建扩展时就为我们提供了非常好的模版文件，更多的时候我们要做的只不过是在相应的方法内填上我们的逻辑，而对于配置方面基本不太需要操心，这一点 还是非常方便的。

就为了扩展这个功能，我已经迫不及待地想用上 iOS 8 了..不论是使用别人开发的扩展还是自己开发方便的扩展，都会让这个世界变得更美好。

原文链接：http://onevcat.com/2014/08/notification-today-widget/

Windows平台分布式架构实践 - 负载均衡

作者：Jesse Liu

概述

最近.NET的世界开始闹腾了，微软官方终于加入到了对.NET跨平台的支持，并且在不久的将来，我们在VS里面写的代码可能就可以通过Mono直接在Linux和Mac上运行。那么大家（开发者和企业）为什么那么的迫切的希望.NET跨平台呢？第一个理由是便宜，淘宝号称4万多台服务器全部运行在Linux，Linux平台下还有免费的MySql,这些都是免费的，这些省下来直接就是利润呀，做企业的成本可以降低又没有任何损失，何乐而 不为呢？第二个理由是在Linux系统下还有很多非常优秀的构架（当然同样也是免费的），分布式缓存Memcached, 大数据处理构架Hadoop等等，这些都为一些大型的分布式系统提供了很好的支撑，当然还有诸如Liniux系统本身的一些安全和网络方面的优势，等等。 所以也难怪大佬们都纷纷不约而同的没有选择.NET。

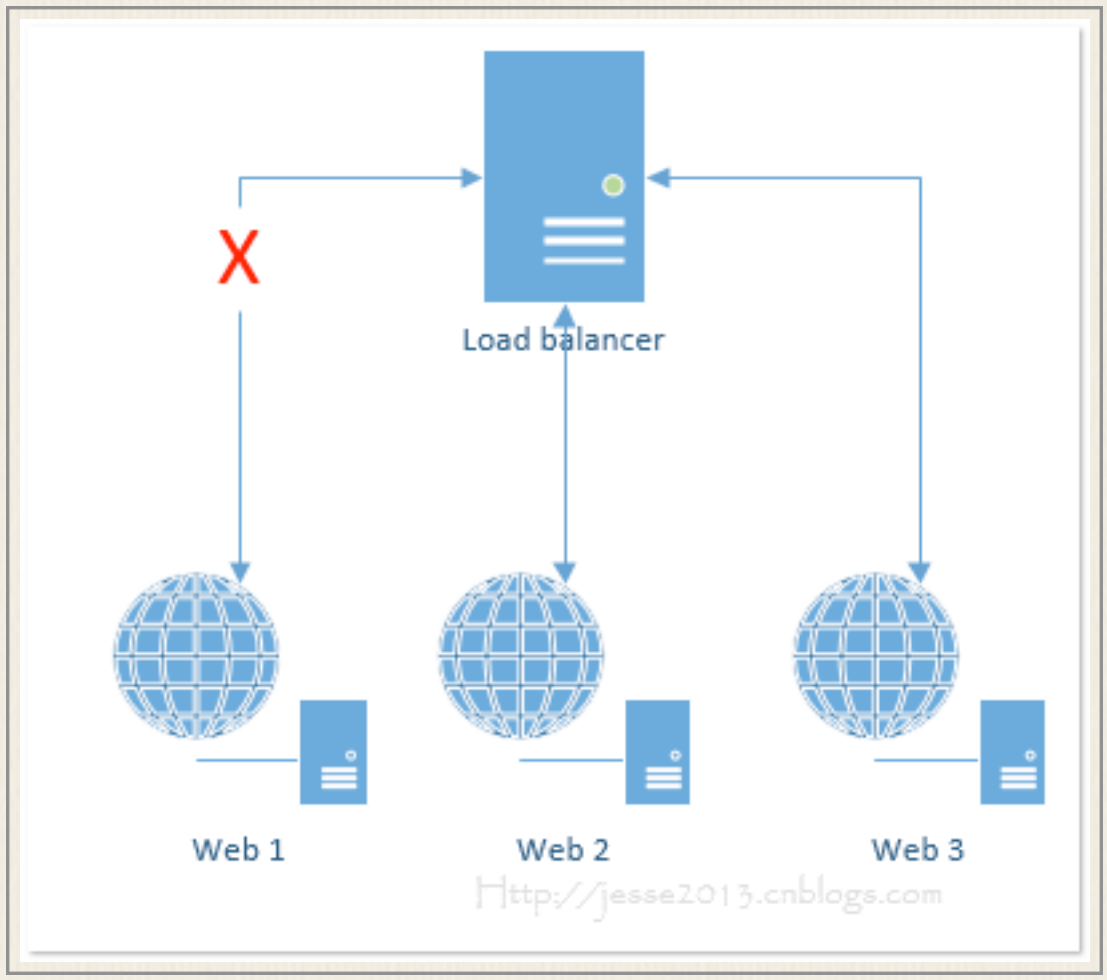
但是如果.NET也支持跨平台之后，那这样的格局可能就要发生变化了。上面所有的优势依然可以保留，并且加上它语法的优越性，以及快速的开发效率等，还是会为其争得一席之地的。

但是，是不是Windows平台下就不能实现这些大型的分布式系统呢？我相信这个问题已经被广泛讨论过，但是至少我没有看到比较清晰的，完整的 案例。带着这些问题，我决定升级我的机器，自己从头到尾在windows平台下搭建一个高可扩展性的分布式网站出来。我经验尚浅，很多的东西还处于摸索阶 段，所以如果有错误，还请大师多多指点。

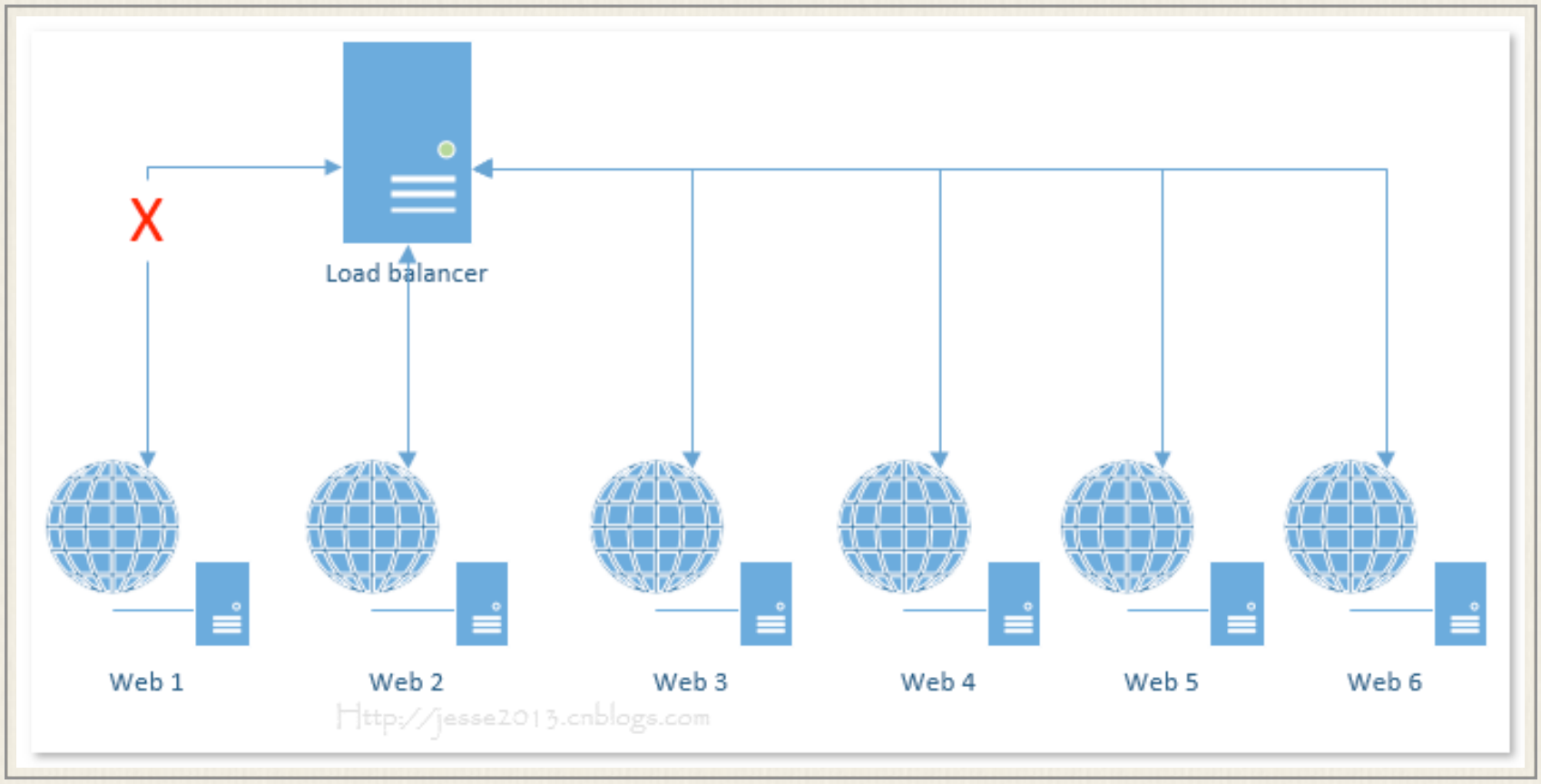
什么是负载均衡

负载均衡可以帮我们解决两个方面的问题，第一个即提高可用性。这里面的可用性主要是从WEB服务器，的角度 来讲的，如果说我们只有一

台Web服务器，而它遇到了某种未知的错误导致IIS无法启动，那么我们的网站就无法访问了，这就是一种比较低的可用性。那么利 用负载均衡，放在我们Web服务器的前面，由它来收集所有的请求，然后转发给我们的Web服务器， 这时候我们就可以添加两台Web服务器，如果其中有一台坏了，至少还有另一台在工作，也不至于导致我们的网络无法访问。



当然，有人可能会问，如果那台Load balancer坏了怎么办？那不是还是访问不了网站么？我们这里讨论的是提高可用性，在做到365天*24小时不间断的服务，需要有另外的组件来支撑，我们留在后面讨论。除了可用性以外，负载均衡还可以帮助我们提高可扩展性，当 然这个可扩展性同样是指的Web服务器层面。从网站性能的角度来讲，好几个程序员花上好几天的时间做了一些优化所带来的效果有时候可能还没有直接加一根内 存条来的快。内存加完了没什么影响，我们还可以换更好的CPU，CPU换完了，我们还可以用固态硬盘，甚至很多公司已经开始直接把数据放到内存中了（注： 具体场景具体对待）。 如果这些都不可以再加了呢？那就再加机器吧，一台服务器可以处理1000个并发，那么两台理论上是2000了，所以这就有了我们的横向扩展。



负责均衡器分发请求的类型

所有的请求首先全部到达Load balancer，再由它转发到具体的Web服务器，转发的方式分为以下几种：

- 轮转调度(Round-robin):最简单的方式，这种方式基本上不能算是负载均衡。第一个请求给web1，下一个给web2，再下一个给web3... 不会考虑某 一个服务器是不是负荷太重等等。
- 基于权重的分配(Weight-based): 可以配置每一台服务器处理请求数据的比例，特别适合那种有某台服务器配置不一样的场景。比如说某台服务器配置特别好，那我们可以让它多处理一些请求。
- 随机（Random): 随机分配。
- 粘性session（Sticky Session): Load balancer会跟踪请求，确保同一个session id的请求都交给同一样服务器。
- 最空闲优先（Least current request）：将最新的请求转发给当前处理请求数量最小的那个服务器。

- 响应时间优先(Response time)：哪台服务器当前响应时间最短就给哪台。

- 用户或URL参数选择(User or URL information)：部分负载均衡器还提供根据一些参数来决定哪台服务器来处理，比如说根据用户信息，地址位置，URL参数，cookie信息等 。

我们还可以根据负载均衡器所使用的技术将它们分为以下几类：

- 反向代理：负载均衡器作为一个代理，同时维持着两个TCP请求，从客户端接收请求，然后将请求转发给相应的 Web 服务器,等Web返回Response的时候是返回给了代理服务器，然后再由代理服务器转交给真正的客户端。这样就会导致有一些功能不可用，比如在WEB 服务器环境查看请求的来源IP实际上成了我们代理服务器的IP等。

- 透明反向代理：和上面的代理服务器一样，只不过WEB服务器从Request中获取到的信息是真正客户端的信息，就是好像没有使用代理一样的。

- 直接服务器返回：通过更改WEB服务器的MAC 地址来实现分发请求，在这种方式下，WEB服务器不会像上面使用代理服务器一样，请求处理完之后是直接返回给客户端的，所有相对于反向代理来说这种方式的性能会更快一些。

- NAT 负载均衡：NAT(Network Address Translation网络地址转换)，将网络包（可以理解成TCP包）中的目标IP地址变成实现要处理这个请求的WEB服务器的地址。

- Microsoft 网络负载均衡：Windows 自带的负载均衡组件，一会我们就用它来做测试。

不使用负载均衡的测试结果

一台独立的服务器

我们可以从一个网站的最初级版本开始说起，最开始的时候我们决定搭建一个网站，但是我们也不知道效果会怎么样，光键是那时候，我们很穷，于 是我们租用了一台托管主机，它可能承担了至少三个或以上的角色：WEB服务器、静态资源服务器，以及数据库服务器。我们可以用

ASP.NET MVC4 + SQL 2008来做一个基本的电子商务网站，基本够用了。但是能够承载多大的访问量呢？下面我们来做一个简单的测试（注意：本文以后本系列所面所有的测试都是在 虚拟机上进行的，忽略网络的因素，以及多台虚拟机同时运行时CPU资源的因素，所以测试结果只是一个参考）。

在我的机器上有一台虚拟机配置如下：

CPU: Intel Core I5- 4570, 3.19GHz,
内存: 4G
硬盘：20G (ShineDisk 固态硬盘)

测试页面没有什么复杂的逻辑，利用ASP.NET MVC4 + Entityframework 6.0 + SQL 2008 + IIS8.5来实现， 我们的页面也只是一个简单的列表页，列出系统里面所有的商品。

Home Controller 代码

```
public class HomeController : Controller
{
    private CarolContext db = new CarolContext();

    0 references
    public ActionResult Index()
    {
        return View(db.Products.ToList());
    }
}
```

Index.cshtml 代码

```
@model IEnumerable<Carol.Web.Models.Product>
@{
    ViewBag.Title = "Home Page";
}
<div class="content">
    @foreach (var p in Model)
    {
        <div class="list-item">
            <ul>
                <li>
                    <img src="@Url.ListingThumb(p.Id,p.ListingThumbName)" />
                </li>
                <li><a>@p.Title</a></li>
                <li>@p.Price</li>
            </ul>
        </div>
    }
</div>
```

在数据库初始化的时候插入500条测试数据

```
public class CarolInitializer : DropCreateDatabaseIfModelChanges<CarolContext>
{
    0 references
    protected override void Seed(CarolContext context)
    {
        var random = new Random(500);
        for (var i = 0; i < 300; i++)
        {
            var product = new Product
            {
                Title = string.Format("Product {0}", i.ToString()),
                Price = random.Next(1000) * i,
            };

            context.Products.Add(product);
            context.SaveChanges();
        }
    }
}
```

连接字符串就使用本地连接就可以了。

```
<connectionStrings>
  <add name="CarolContext"
connectionString="Server=localhost;database=carol;trusted_connection=tr
ue" providerName="System.Data.SqlClient" />
</connectionStrings>
```

我们使用的轻量级的ab来做压力测试，如果不熟悉ab的可以点这里，下面是测试的结果：

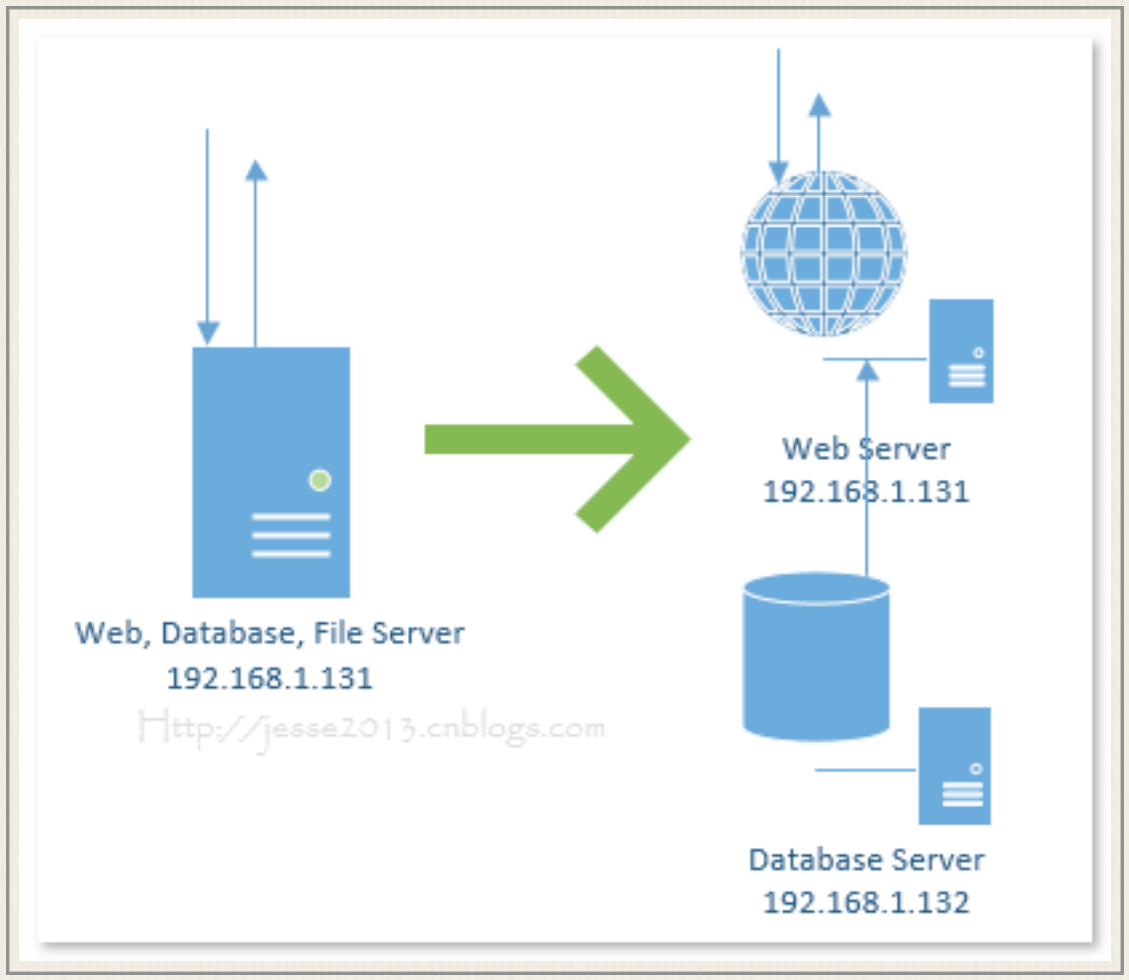
```
ab -n1000 -c100 http://192.168.1.131
```

总请求数	并发用户数	每秒处理请求数	每请求处理时间 (ms)
1000	10	113.98	87.732
1000	100	115.71	864.216
2000	100	119.90	833.96
1000	200	122.96	1583.53
2000	200	118.38	1576.53
1000	300	128.91	2327.259
1000	400	127.53	3136.637

通过测试发现，我们这单个服务器的吞吐率接近在110~130之间，而一旦并发数达到200以后，每个请求的处理时间就达到1.5s多了，400个并 发用户的时候每个请求要花上3s多的时间。如果在真实的网络环境中可能会更差。由此我们可以得出我们这个服务器可能最大支持120人左右同时访问。

WEB服务器与数据库服务器分离

现在我们来做一个花费不是很大，又空间做的扩展，也不需要改任何架构，我们只是再加一台专门的数据库服务器。



下面我们再来看一下测试结果：

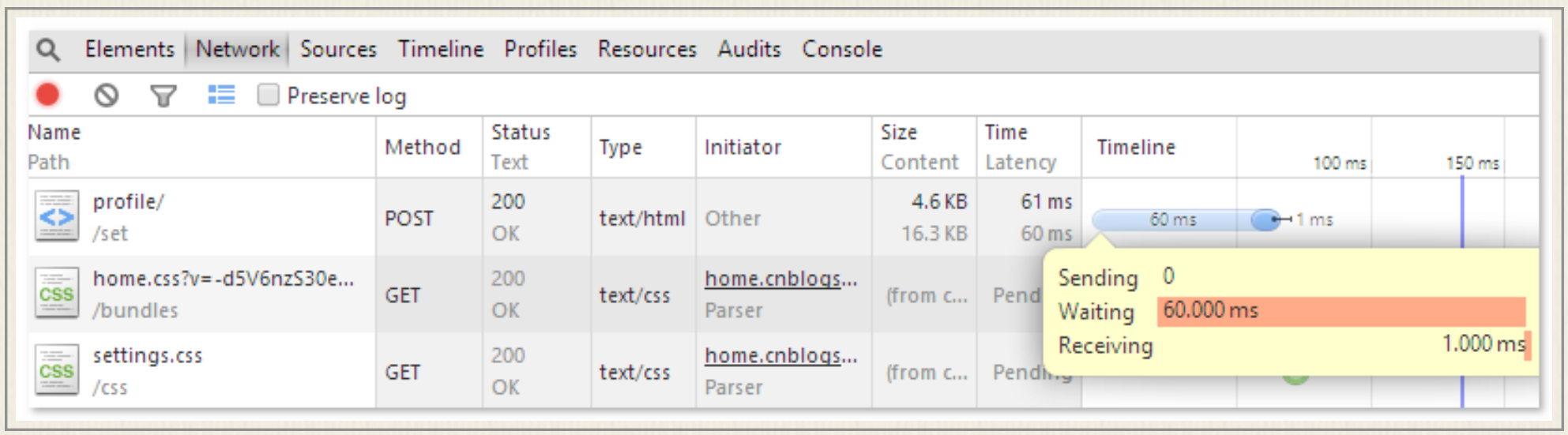
总请求数	并发用户数	每秒处理请求数	每请求处理时间 (ms)
1000	10	145.62	68.670
1000	100	150.17	665.934
2000	100	151.94	658.146
1000	200	156.59	1277.205
2000	200	160.48	1246.286
1000	300	150.61	1991.916
1000	400	154.88	2582.637

大家可以看到，这里我们的吞吐率(每秒处理请求数已经提升到了150左右)，并发处理能力提升了50%，并且300和400并发的时候响应时间也比上面的架构要好一些。

使用负载均衡的测试结果

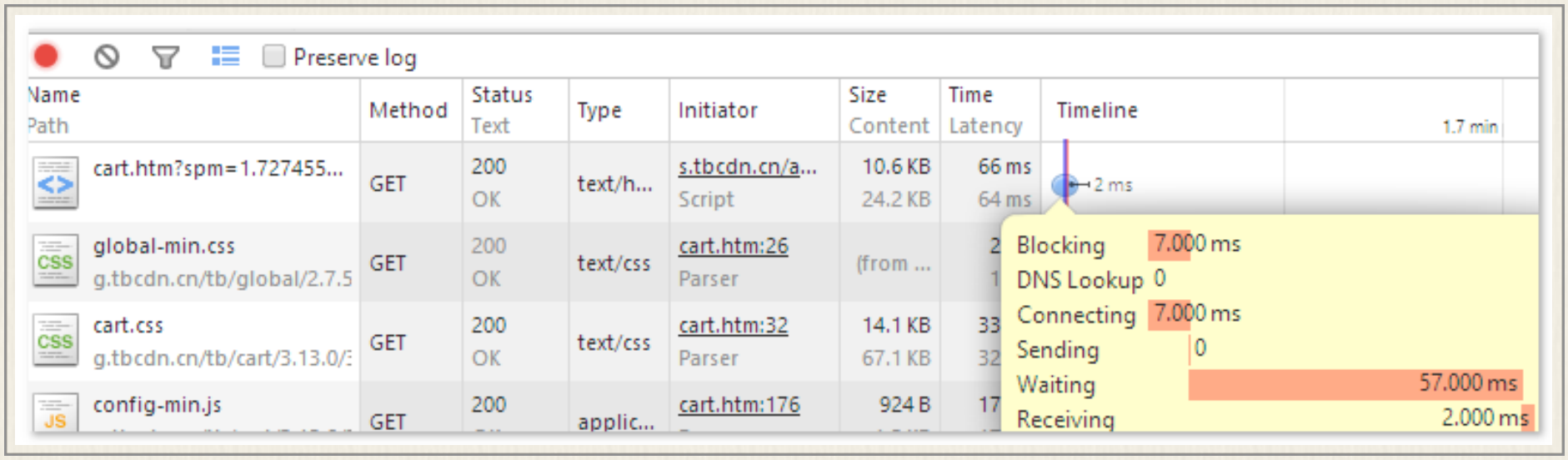
安装网络负载均衡（NLB)

上面我们一台独立的Web服务器和一台独立的数据库服务器的组合已经可以处理150左右的并发了，现在我们假想一下如果网站的的知名度越来越 大，如果同时有400个用户来访问怎么办？ 从上面的图中我们可以看到400个并发的时候服务器的处理时间为2582.637ms（实现上这是拿到响应的时间，因为我们是一台机器上的不同虚拟机，我 是在主机上做测试，所以我们就忽略网络传输的时间，假设这个就是我们的服务器处理时间），这个服务器响应时间也就是我们通过F12->网络 中看到 的等待时间 。



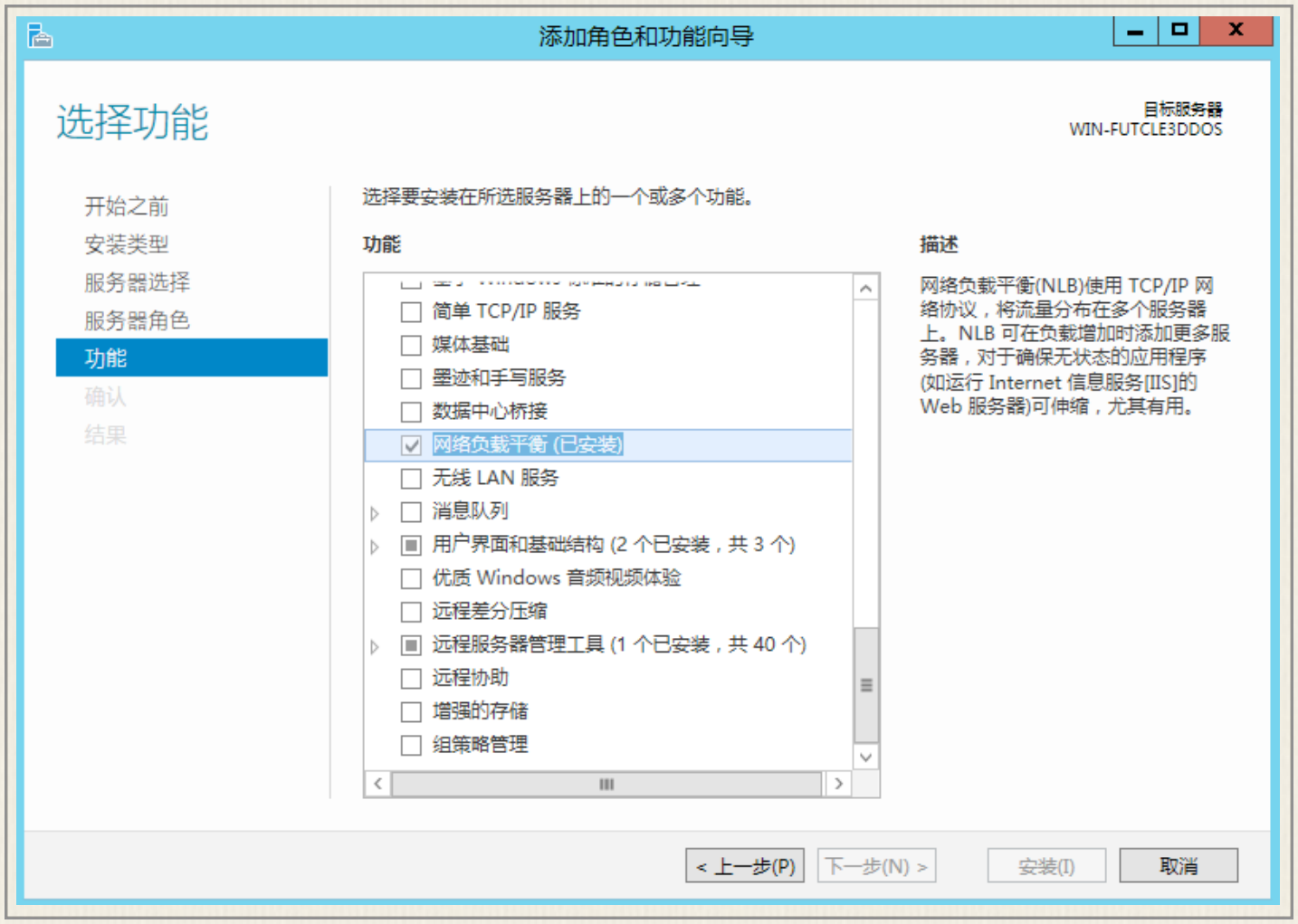
页面什么时候能拿到这个响应还要加上网络传输的时间，也就是Receiving。1ms的传输时间堪称神速啊！我家用的长城宽带10M，总是早上网络出奇的好，一到晚上就挂掉了，还有可能就是你们现在都没有上博客园 :)

用户体验黄金法则之一: 网站加载时间 = 用户体验，别说3S，可能等个2S你页面还不出来，用户准备离开了，下面是淘宝购物车页面的加载时间 。



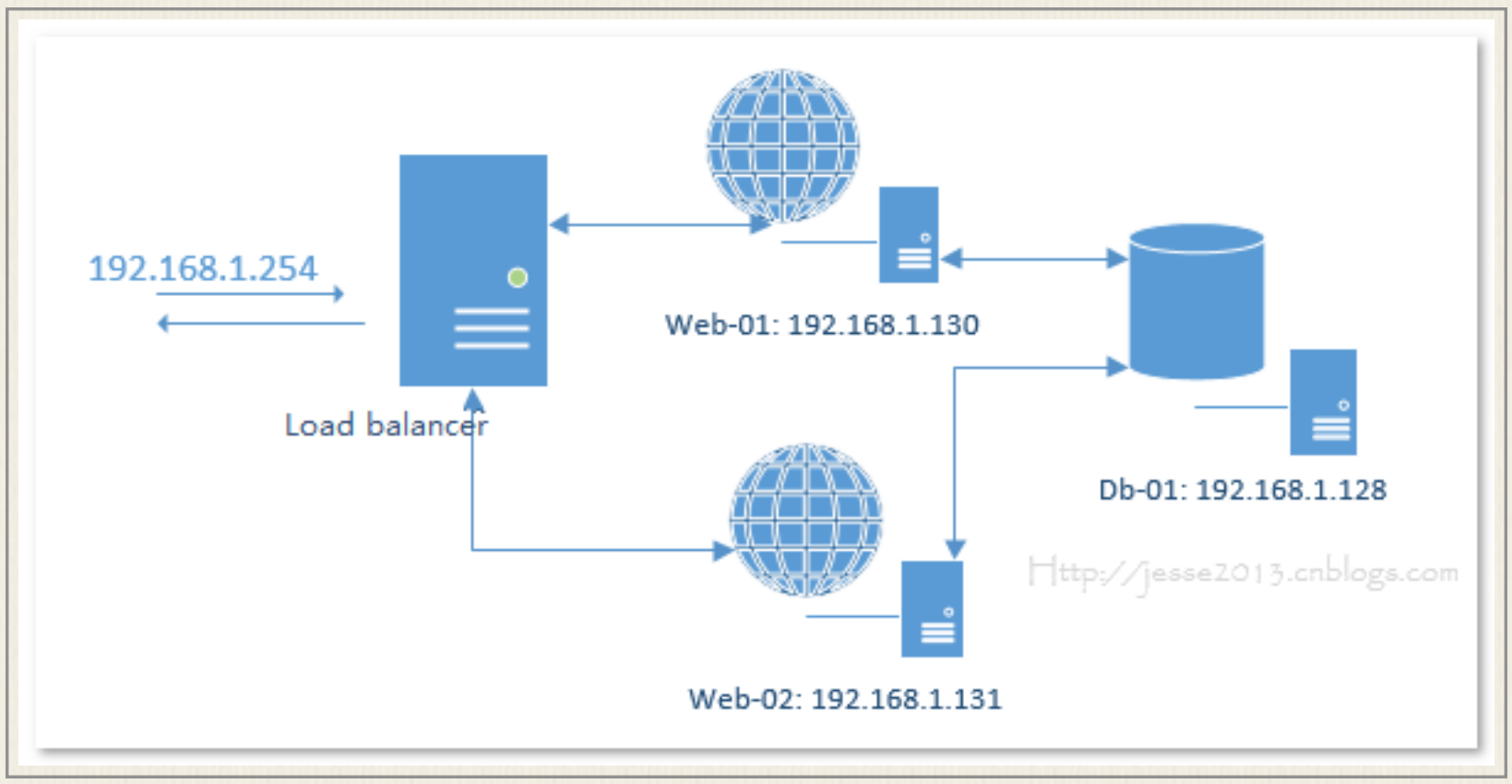
国内很多大型的网站的响应时间基本上都控制在100ms以内，基本达到那种一输入地址敲回车，眨眼之间页面就出来了。当然这并不是光有个负载均衡加 几台web服务器就能解决的，我们后来再来一步一步的实践下去。话说回来，我们上面的测试结果基本上只有并发为10的时候响应时间是在100ms以内的， 看来我们还有很长的一段路要走啊。

正所谓“最好的架构是进化而来的，而不是设计出来的”，面对我们现在的瓶颈暂时通过负载均衡添加多台Web服务器就可以了。我们上面讲到负载均衡器类型的时候有一种 Microsoft负载均衡，我们可以很轻松的通过服务器管理器来将这些组件安装到我们的服务器中。 安装我们就不讲了，就是通过服务器管理-> 添加角色和功能->在功能中选择“网络负载均衡” 然后安装就可以了。



注意：图中的Load balancer实际上是不存在的，因为只要我们2台Web服务器安装了网络负载平衡组件，在其中任意一台上建立群集就可以了，图是为了方便大家理解。

这样的话我们的服务器架构就成了下面这个样子：

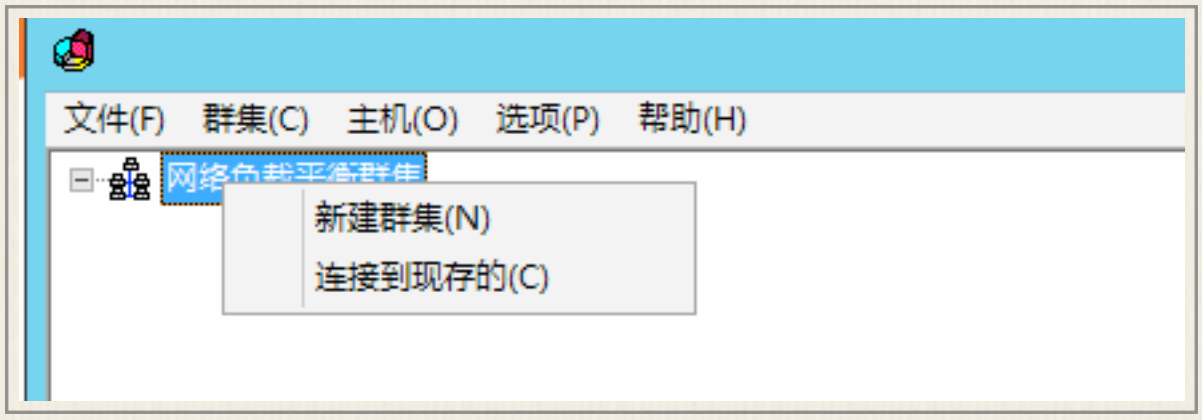


192.168.1.254 就是我们暴露的外部IP地址，访问192.168.1.254的请求就会转发给后面的两台WEB服务器。

配置网络负载均衡

在我们为上面2台WEB服务器安装NLB之后，我们在其中任意一台上来新建群集，然后将另外一台加入到这个群集中即可，我们就在web-01(192.168.1.130)上来新建这个群集。在建立群集之前，我们要确保这2台服务器都是使用的静态IP，否则无法将他们加入到群集中。

- 在web-01(192.168.1.130)上从管理工具中打开 网络负载均衡器
- 右击“网络负载平衡群集”，选择“新建群集”



在“新群集：连接”窗口中将 192.168.1.130添加为主机，点击下一步

进入 “新群集：主机参数”，直接下一步

进入 “新群集：群集IP地址”， 添加窗口中的“添加” 将192.168.1.254 添加到窗口中然后点击下一步

新群集: 群集 IP 地址

群集的
地址，

IP

群集 I

IP 地

添加(A)..1 | 编辑(E)... | 删除(R)

< 上一步(B) | 下一步(N) > | 取消 | 帮助

添加 IP 地址

添加 IPv4 地址(A):

IPv4 地址(I): 192 . 168 . 1 . 254

子网掩码(S): 255 . 255 . 255 . 0

添加 IPv6 地址(D):

IPv6 地址(P):

生成 IPv6 地址(G):

链接-本地(L) 站点-本地(T) 全局(O)

确定 2 | 取消

新群集: 群集 IP 地址

群集的每个成员共享群集 IP 地址以进行负载平衡。将所列出的第一个 IP 地址视为主群集 IP 地址，并用于群集检测信号。

群集 IP 地址(C):

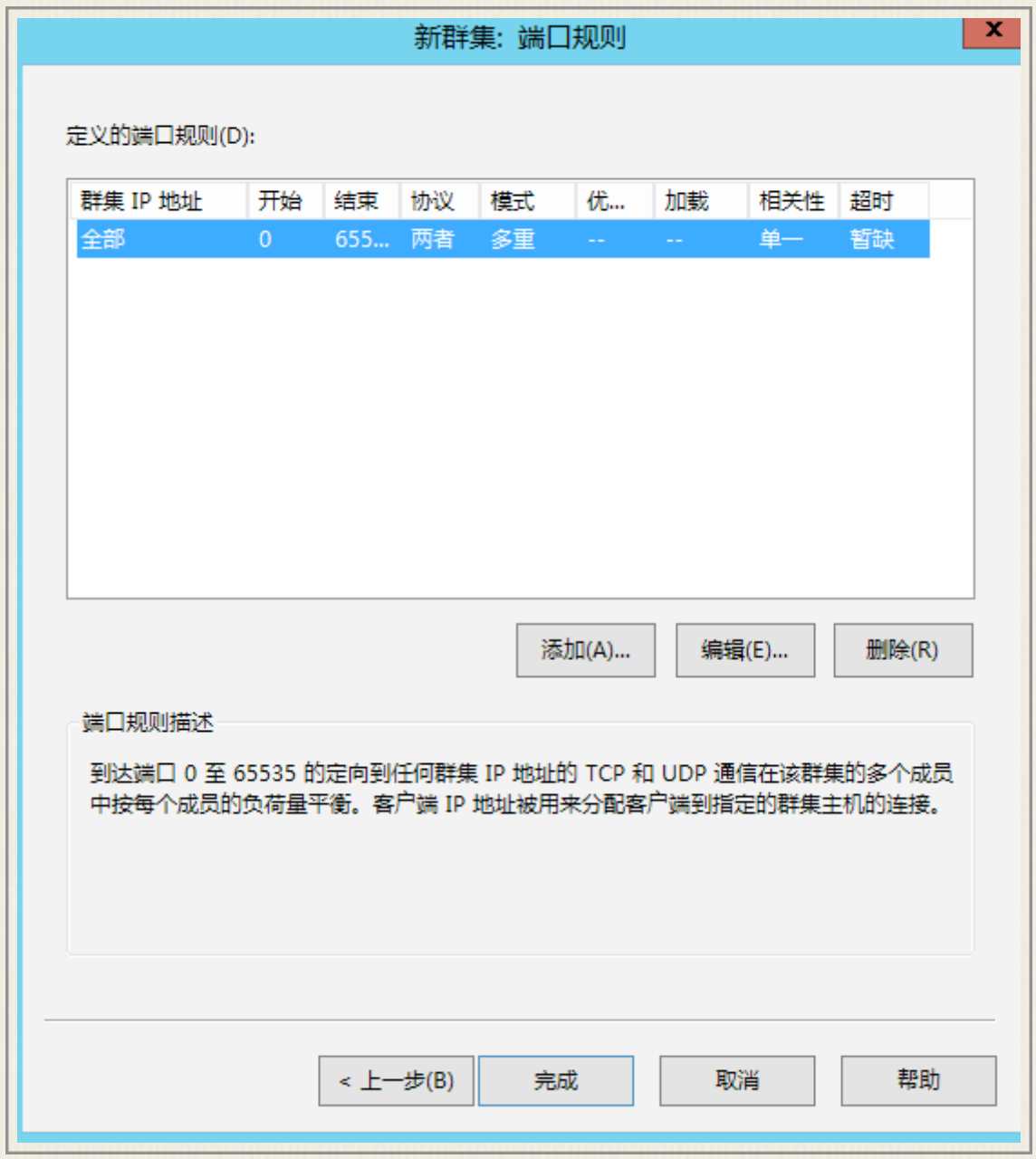
IP 地址	子网掩码
192.168.1.254	255.255.255.0

添加(A)... | 编辑(E)... | 删除(R)

< 上一步(B) | 下一步(N) > | 取消 | 帮助

进入“新群集：群集参数”，选择“多播”然后点击下一步

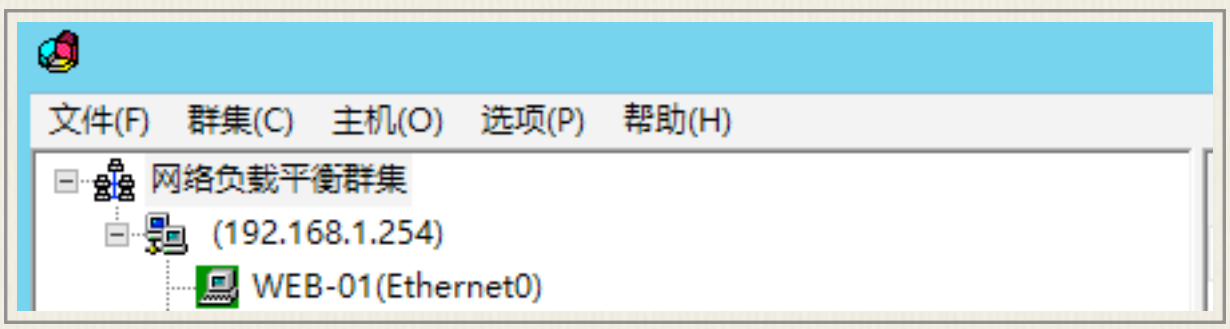
进入“新群集：端口规则”，选中全部，然后点击编辑



将端口范围改成 80~80，协议选“TCP”，相关性选“无”

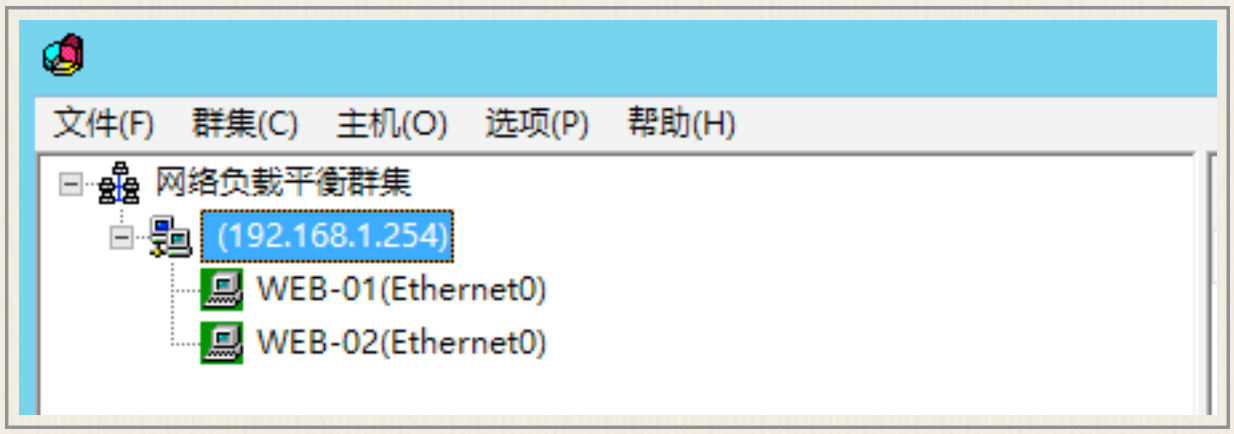
点击确定回到主窗口，然后点击完成。

通过上面的步骤，我们已经建立了一个群集，并且将web-01加入到了群集中，我们还需要手动将web-02也加入到群集中。



在群集(192.168.1.254)上右键点击“添加主机到群集”

在“将主机添加到群集：连接”窗口中的 主机中输入192.168.1.131然后后面一下点下一步即可。



现在我们就可以到我们的真实机器上去访问192.168.1.254了，也就是说马上我们就进入测试环节了。

测试结果

本文中所有的测试结果都没有取第一次的结果，EF也需要预热，同样的查询在EF中也是有缓存的，所以第一次的结果会与后面的测试结果有很大的区别，后面的几次（在相同参数下）基本差别就不大了。

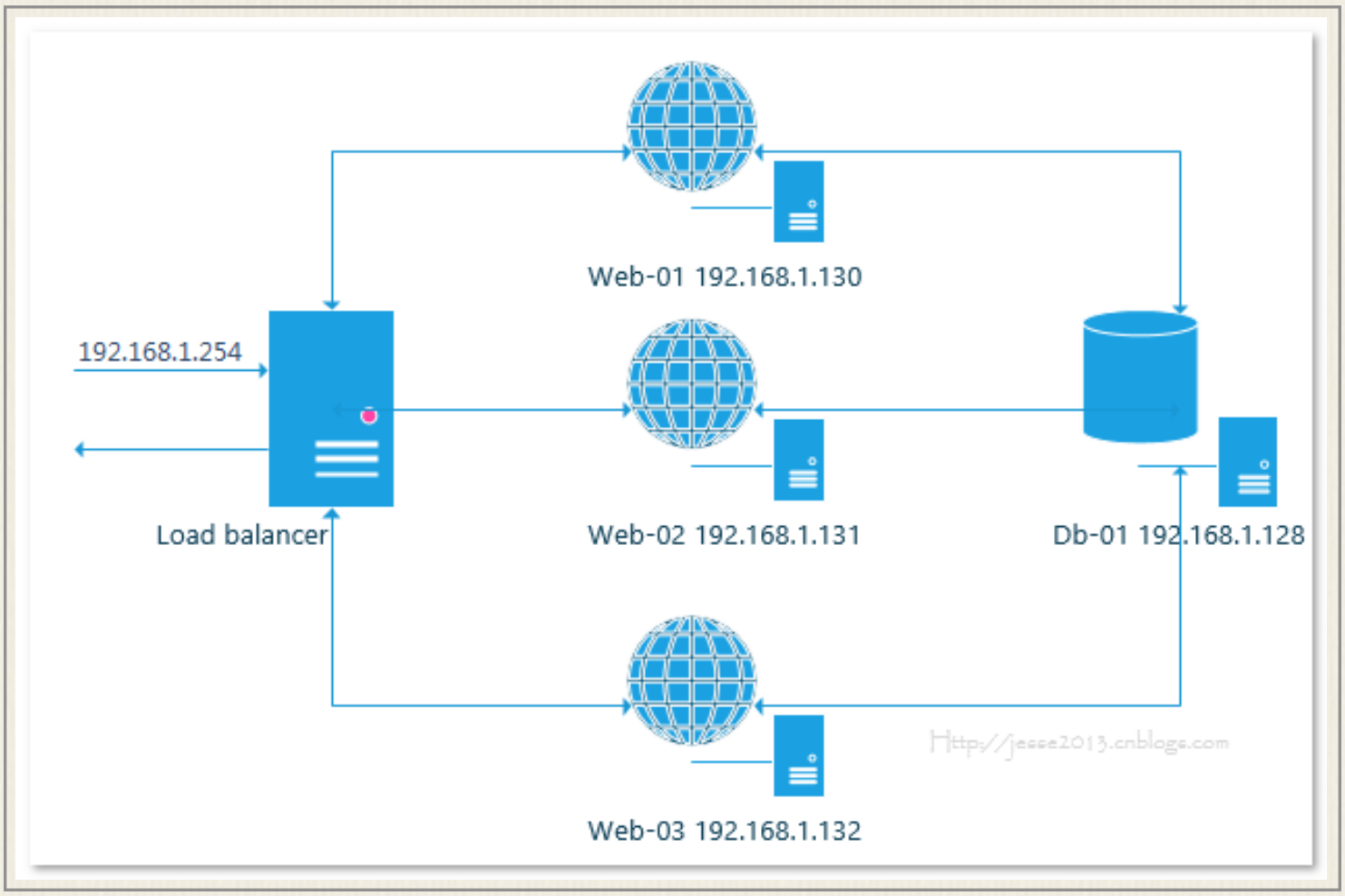
总请求数	并发用户数	每秒处理请求数	每请求处理时间 (ms)
1000	10	227.76	43.907
1000	100	205.75	486.026
2000	100	230.22	434.365
1000	200	221.89	901.336
2000	200	263.46	759.141
1000	300	206.87	1450.194
1000	400	207.6	1931.764

可以看到现在我们的吞吐率大概平均在230左右，与一台WEB服务器+一台DB服务器相比，处理能力又提高了50%，为什么不是100%呢？一台WEB服务器能处理150的并发，那两台应该是300才对呀？我能够想到以下原因：

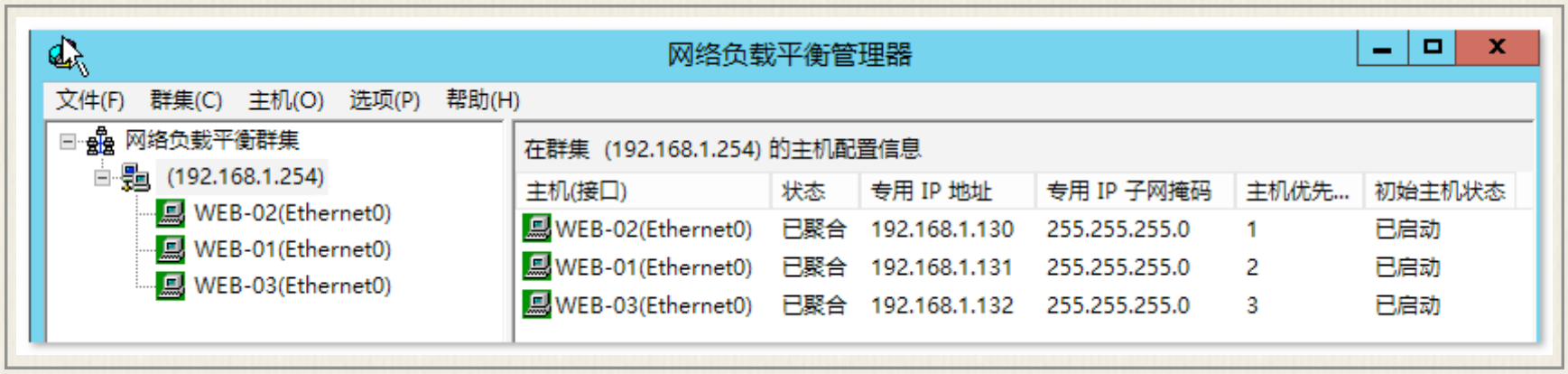
1. 我们的数据库服务器只有一台，数据库的处理能力提不上去最终影响WEB服务器的处理能力

2. 我们采用的是虚拟机，并非实际的机器，他们实际上是共用CPU，不知道在这种情况下对测试结果会不会有影响（虚拟化专家可以分享一下）。

为了验证一下，我再扩展了一台WEB服务器，我们使用3台WEB服务器+1台DB服务器看看是什么效果。



我们新建一台虚拟机web-03，然后将它也加入到我们的群集中。

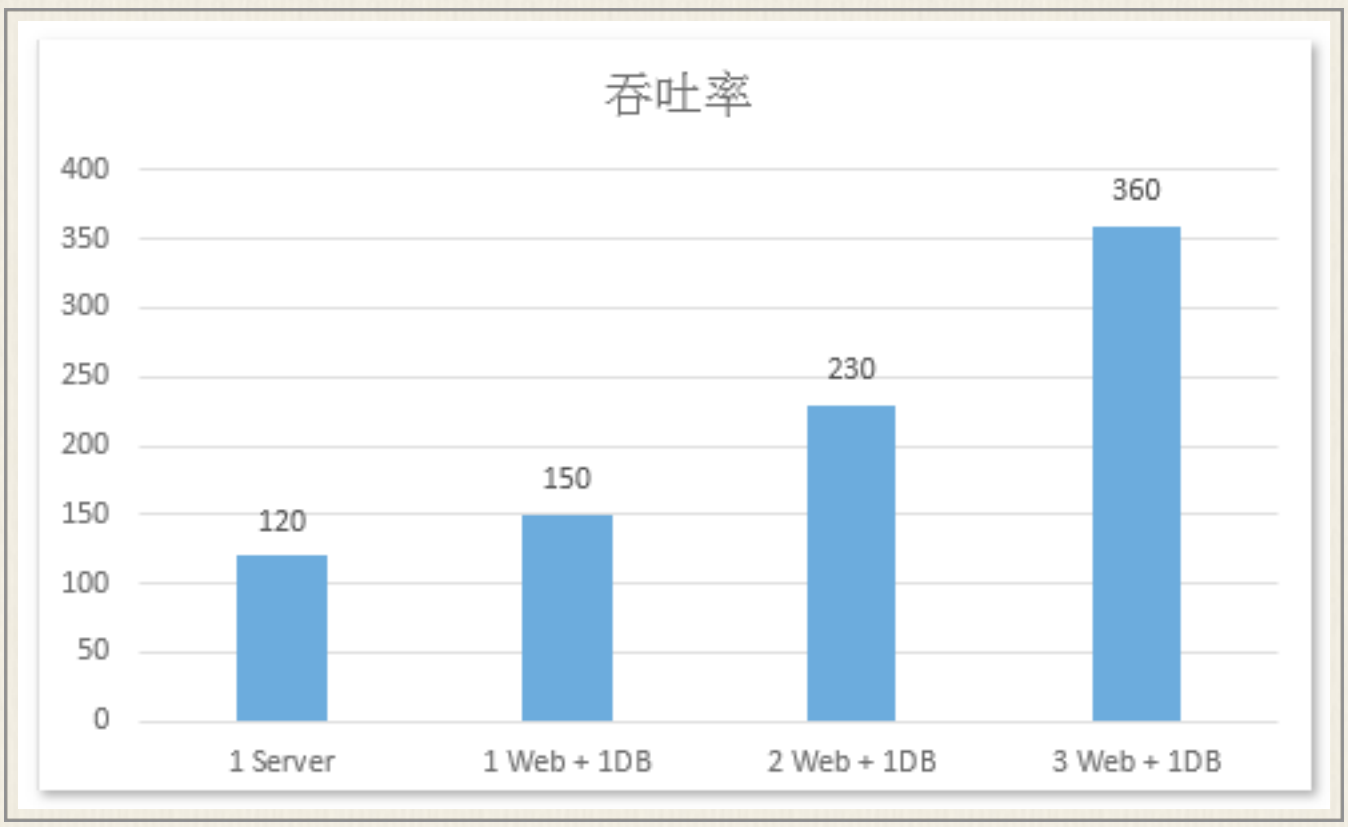


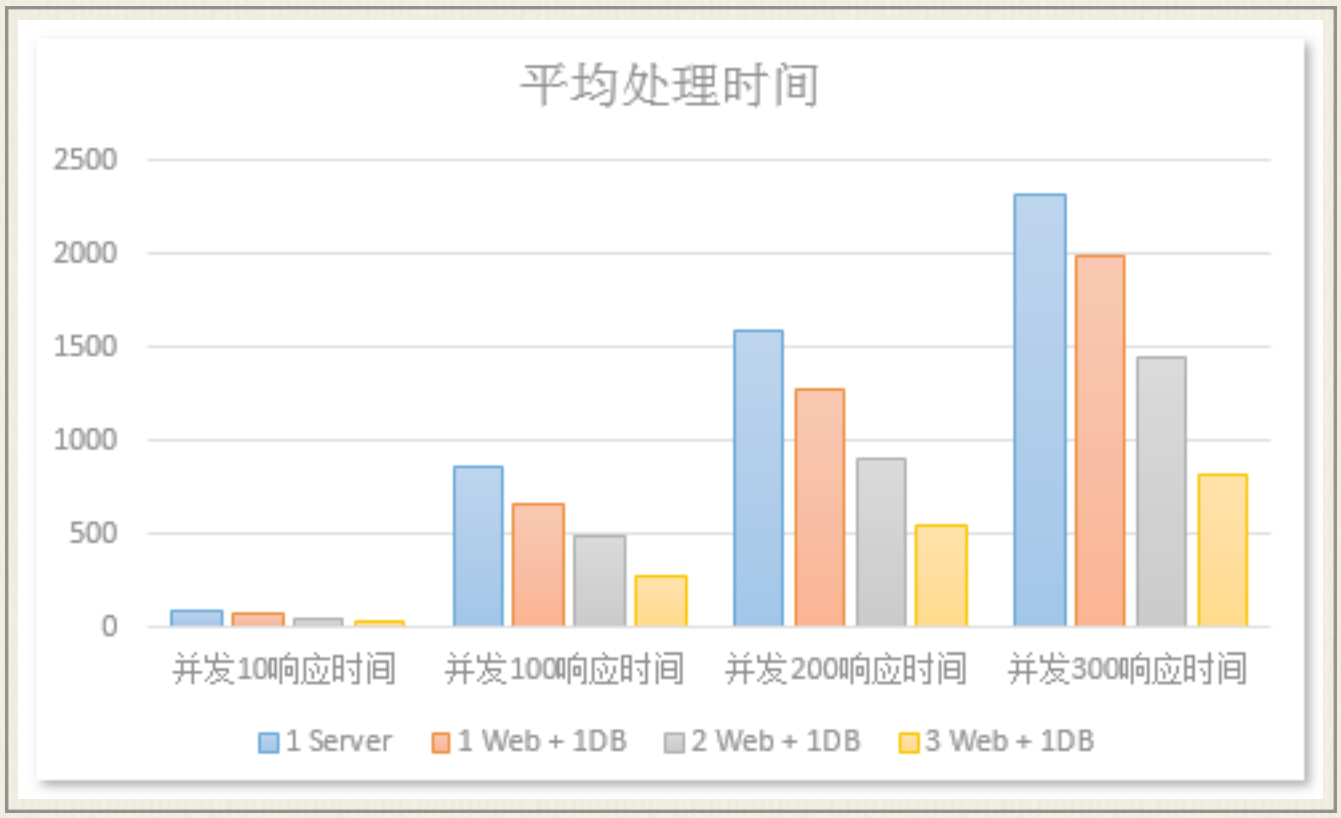
测试开始!

总请求数	并发用户数	每秒处理请求数	每请求处理时间 (ms)
1000	10	335.22	29.831
1000	100	366.31	272.994
2000	100	373.50	267.740
1000	200	367.39	544.384
2000	200	376.24	531.576
1000	300	369.29	812.375
1000	400	364.04	1098.779

在加入第三台WEB服务器之后，我们的吞吐率（每秒处理请求数）再次得到提升从230升至360，并发处理能力再次提升56%，并且大家可以看到，400并发以下的平均每请求处理时间都在1s以内，可喜可贺啊!

最后上两图让大家更直观的看一下这些性能的变化:





以上数据均来自本人机器上的测试，虚拟机全部采用与第一台服务器同样的配置。

小结

在网站架构的不断演变中，负载均衡起着非常重要的位置，不仅仅为我们提升可靠性和可扩展性，有一些比较强大的硬件设备还能提供缓存，以及 session机制。今天我们用到的负载均衡是Windows Server自带的一个组件，它是最简单实现负载均衡的方式，但是功能不是特别完善，而且一旦NLB本身发生错误那么将导致所有的网站都不能访问，我们后 面就来通过引入APR(Application Request Router)来解决这个问题，想要真正了解大型网站的架构实现，而不是仅仅知道负载均衡，分布式缓存，数据库分离这些名词么？那就来跟我一起学习吧！另 外我们今天只是用一个简单的页面做了压力测试，只有读数据的操作，还没有写的操作，也没有任何复杂的事务，但是别担心，我们一步一步来 :) 。

原文链接：http://www.cnblogs.com/jesse2013/p/dlws-loadbalancer.html

技术人攻略访谈三十二| 清风：豆瓣神组小组长，日式萌神程序员

作者：Gracia

导语：本期采访对象清风， 前豆瓣技术总监。豆瓣以小清新风格，和独特的工程师文化在国内技术圈独树一帜。过去四年半里，清风亲身经历了这家创业公司的高速发展，从30多人小团队， 成长为近500人的成熟企业。这期间，他主导了一项重要变革：用开源项目运作方式打造出豆瓣内部代码协作平台CODE，推动开发流程成功从SVN转向 Git。版本系统迁移是牵一发而动全身的大工程，涉及围绕SVN的整个生态系统迁移，工程师技能转型，以及管理流程及文化适应。其难度和工作量巨大，即使 是豆瓣这样极客范儿的公司，也花了两年时间才完成。

诞生于开源世界的版本控制系统Git，由Linux之父Linus Torvalds亲自设计和开发，以实现对庞大复杂的Linux内核项目的高效管理。这个分布式版本控制系统具有强大的分支管理能力，无须操心数据备份， 并能有效避免因代码的频繁提交而中断工作。Git高效的协作模式带来开发效率的极大提升，以及团队成员之间的有效沟通。凭借这些明显强于SVN的特 点，Git迅速在开源界流行开来。

基于Git的代码托管平台GitHub应运而生，不仅有Eclipse、jQuery、Ruby在内众多知名开源项目迁移至GitHub，并且孕育 出BootStrap、Node.js、PhoneGap、Docker等大量明星项目。新工具普及带来生产力解放，开源世界的大门从此敞开，赋予普通人 自由使用和修改代码的权利。协同成本大大降低，任何人都可以Fork代码、创建分支，用Pull request方式参与开源项目的贡献。这场代码民主化运动，是去中心化的互联网力量对核心智力生产流程的一次渗透，借助社交化传播渠道，让每个优秀创 意，都能带来整个领域的机会提升。

清风说自己身体里流着创业的血液，凡事喜欢折腾，而且都做得有模有样：在豆瓣白手起家倒腾出一堆新部门，参与国内最早的比特币项目，就连

玩豆瓣小组，也能玩出一个30多万粉丝的“吃喝玩乐在北京”（什么！真的没有听说过！那你玩陌陌吗？）。在以贴标签文化著称的豆瓣，吃喝组组长清风喜欢在自己“白羊座”的标签之前加上“纯洁”这个形容词，同事的标 签则是“表面纯洁的白羊座”、“萌神”、“卡卡西”。清风是野路子出身的非典型程序员，早在18岁那年就见识了互联网泡沫的疯狂和幻灭，于是在他典型的白羊座式的萌里，还看得到过尽千帆。

技术人攻略：你在豆瓣工作的时候做了开源代码协作平台CODE，能讲讲当时做这件事的背景吗?

2009年我离开工作了四年半的新浪，想去一家技术范儿特别重的公司，其实心里只有一个目标，就是去豆瓣。作为豆瓣的深度用户，我非常喜欢这个 网站。05年在Python User Group聚会上，听阿北介绍过豆瓣的技术架构，豆瓣是国内第一家，也是惟一一家大规模用Python搭建主体业务的公司，我希望能在Python上钻得 更深入。加入豆瓣时，我刚好是它的第36名员工。

我在豆瓣待了四年半，眼睁睁看着它从30多人变成了400、500人的公司（技术团队超过300人）。发展的每一个阶段，管理方式都会有相应的 变化。在豆瓣的第一年，技术部总共才十来个工程师，当时采用接单式工作法：所有工程师在一个池子里，任何产品要做新功能，就看哪个工程师有时间，整个管理 很轻。

公司接近100人规模时，这种方式的弊端出来了，工程师和产品没有关联，缺乏归属感，沟通效率极低。于是改用产品线管理方式，每条产品线配备一 个包括产品、运营、开发和测试在内的全编制团队。同时期还诞生了几个公共部门，如Anti-Spam、商务支持、公共服务、平台组和算法组。

组织结构的变化影响着开发流程和工程师文化。归属感的问题解决了，但技术的横向交流变少，时间长了不免出现重复造轮子的问题。

为了解决这个问题，一开始是想办法把工程师聚到一起，例如豆瓣Happy Day活动：每个季度腾出一天工作日，让所有工程师放下手里的工作，一起来写代码。这个活动搞了好几届，分了不同的主题，包括类似ACM的算法大 赛，Hackson大赛，还攒过一次3D打印机。这个活动要求所有人停下工作，成本非常高，缺乏持续性。

后期我们希望用一套流程和工具来帮助工程师沟通。豆瓣的工程师Geek范儿很足，发明一套流程他们绝对不会用，于是自然而然想到了 GitHub。GitHub在开源界广受欢迎，开源文化也和豆瓣的一贯气质最为符合。GitHub严格意义上不只是个代码托管平台，而是一个程序员社交平 台，它那套机制之所以运转得好，真正原因是让工程师实现了顺畅的交流。按豆瓣工程师人数算，Github企业版的服务一年下来得花50万人民币，关键当时它很不稳定。于是只好下决心自己开发一套，这就有了后来的豆瓣CODE。

技术人攻略：做豆瓣CODE的过程中，最大的困难在什么地方?

从立项到整个开发流程都转到CODE上，前后一共用了两年。需要完成两件事，一是搭建一个类似于GitHub的代码平台，二是把整个开发流程从SVN转到Git。回想起来，这个过程真不是特别容易。

第一个比较大的困难是没有人手。我被耿老（耿新跃）和洪教授（洪强宁）赶鸭子上架，做了这个项目的Leader，但却是个光杆司令，手下一个专 职的人也没有。2012年情人节，我写下了CODE的第一行代码，和洪教授带着两个实习生，用了一个星期，基于Python开源问题跟踪系统Trac改出 了一个demo。界面巨丑无比，但功能都有了，于是咬着牙开始用。

推广第一步，先从自己的产品线开始用。我带的产品线很多，例如Anti-Spam，商务开发、东西产品线，另外洪教授的平台组也在用。当时的做法是边做边迁移，边加功能，完全是配合着需求，边走边造轮子的过程。

那段时间我几乎把50%的精力都放在CODE上，贡献了大量代码。但一个人力量有限，不可能实现所有功能，于是开始忽悠更多同事参与开发。GitHub当时正好拿了一笔融资，估值到了7亿，我于是见人就画饼：你们想参与一个价值7亿美金的项目吗？当然并不是完全山寨GitHub。最后统计共有48人参与了项目，完全按着开源软件的节奏，大家见缝插针，凭着热情和爱好，把CODE成功地做了出来。

做这件事的第二大难点，是统一大家的思想。SVN转向Git不仅涉及使用习惯的变化，更重要的是，围绕SVN构建的整个生态系统都需要随之而变。豆瓣很早就开始用SVN，并围绕它打造了很多工具，包括上线系统、持续集成系统、任务管理系统都和SVN绑定，相当于所有这些代码都要重写，涉及巨大 的工作量。

Git的学习曲线比SVN高很多，要想上手，除了需要掌握百条左右的Git命令，还得了解GitHub这套流程规范。即使在豆瓣这样Geek范 儿的公司，也并不是所有人都参与过开源项目的开发，熟知Pull Request那套流程。一些积极性不高的员工难免会产生抵触心理，这时候就需要借助公司的力量，从上到下推动这件事。一方面在产品线内部培养积极分子， 另一方面提供相关的培训，再有就是采取一定强制措施，从公司层面强制规定使用Git。

技术人攻略：豆瓣的工程师文化，对做成CODE这件事有多大帮助?

国内号称有工程师文化和氛围的公司，豆瓣真得算一个。在豆瓣，工程师文化不是形而上的概念，而是落实到通过工具，而不是通过制度去解决问题的行为方式。创始人阿北是技术背景，包括洪教授在内的头几个员工，有非常优秀的工程意识，他们一开始树立的高度，决定了整个公司延续下来的文化。

我到现在都还记得，刚进豆瓣的第一天，我的工作不是开发产品，而是和同事做一个方便前端工作的框架。豆瓣很早就已经有一套上线系统，即使当时公司的规模还那么小，而且即使在那么早期，大家都会主动写单元测试。

CODE之所以能做成，是全豆瓣所有人的功劳。几乎每个人都愿意去做过程改进这件事，只要环节中有浪费，有不爽的地方，大家就会想办法解决。各 个技术公司都在强调工程师文化，但要真正把这件事做起来，不能空谈，需要有一些形式和具体可执行的东西。比如每周的技术分享、促进工程师沟通的工具，甚至 晚上的吉他班。什么是工程师文化？代码才是最重要的！一家不重视代码的公司何谈工程师文化。

技术人攻略：根据你的观察，国内公司和程序员对Git的接受程度如何?

国内的Geek程序员已经开始用GitHub，但仍然有大量的人从没接触过开源。找我做咨询的一些公司，程序员还在用Windows做开发，即 使在豆瓣这样的Geek公司，到后期也至少有一半的技术人没有参与开源。所以开源这件事，在中国还仅限于小部分人。我和国内互联网及传统软件公司的人都交 流过，豆瓣转到Git尚且用了整整两年，对于那些规模大得多，Geek比例低得多的公司，推广难度更不可估量。

国内的互联网公司很缺人，但缺的是A类和B类人才，这类人才不可能通过大学教育教出来，只能靠开源文化的传承去熏陶。GitCafe、技术社 区，以及国内做黑客马拉松的公司，大家都在做一件事：通过这些活动和产品，让更多人接触到开源文化，帮助国内工程师尽快成长，变成那种更加Geek的程序 员，这将是个很漫长的过程。

2003年我加入新浪，跟着老黄（黄冬）推SVN，那时候还没有公司用版本控制工具，都是每天复制一份文件夹，用FTP上传到服务器。那会儿大家的疑问是：为什么要用版本控制工具？十年过去了，新的问题变成：应该要用版本控制工具，可为什么要用Git？这个变化在我看来已经是进步了。再举个例 子，原来很少有公司写单元测试，现在这个比例虽然仍不高，但起码有人愿意开始写了。过去京东、大众点评都是基于.NET架构，而现在都在转向开源架构，阿 里这样的公司更是为开源做了很多贡献，去IOE就是阿里发起的。这些在我看来都是好现象。

技术人攻略：你经历过豆瓣比较完整的发展过程，解决了开发流程问题之后，还遇到哪些问题？

2012年业界开始兴起O2O概念，豆瓣那会儿的整体方向是接点地气，所以也开始涉足这个领域，例如卖电影票，做电子书，参与音乐节等。线上业务一旦和线下挂钩，就得铺人。例如卖电影票得做交易系统，对接所有的电影院平台和取票机，每家的平台都不一样，业务复杂度很高。除了技术，还得铺大量的运 营人员，给运营同事开发管理后台、便利工具等等，都需要人，于是公司开始大规模扩张。

扩张期招来的人，不可能都是Geek，校招又进来了很多小白。新人直接下放产品线后，出现了很多问题。于是设计了一个新人训练营，在一个月培训期里，做集中式的工具讲授和开源文化的普及，效果好了很多。

回头看豆瓣这两年的战略发展，移动客户端和O2O这两波浪潮都没抓住。这两个领域有个共同点，恰巧是豆瓣不太擅长的地方。PC互联网时代，不用 投钱推广，通过SEO就能从搜索引擎获取大量流量。而移动互联网时代，免费流量没了，需要铺人做渠道，才能把下载量做上去。O2O也类似，不可能借助免费 的渠道起来，必须要铺人、铺渠道。

时代变了，但我们没有顺应去变，所以在这两个关键点上豆瓣都做得不太好。这其实是个后知后觉的事，做的时候对情况估计不足，比如豆瓣电影

的口碑 很好，本想着卖电影票是顺利成章的事，但真正推起来的时候，才发现影院并不买帐。虽然豆瓣在扩张，但总体人数和团购网站比还是小得多，在地推的时候并不占 优势。

技术人攻略：听说你比较早就进了互联网行业，还经历过2000年互联网泡沫，当时是什么情况?

我出生在北京，接触电脑非常早，小时候就经常去中关村买软盘，眼睁睁看着这条街一点点变化，从这里感受到IT前沿的力量。中关村诞生了许多牛 人，刘强东当初就在那开柜台，爱国者的冯军也是从卖键盘、鼠标这样的小生意起家，还有王志东、王江民这样的传奇人物，我们都听着这些人的传说长大。

高一开始上网之后，不知怎么就莫名其妙买了本杂志，跟同学一起学HTML。当时正好有一些海归回国创业，依靠HTML这个“高级技能”，我们接 了些兼职工作，一天能挣150块钱。当时小，能挣这么多钱非常开心，于是决定退学。结果自然是没退成，只好偷摸着搞网站。我们喜欢玩游戏，手上也有很多相 关资源，就做了个游戏资讯网站。

高考结束当天，我和同学就在潘家园租了一房，开始折腾自己的事。当时主要的收入来源，是给大网站的游戏版块提供资讯和下载资源。没想到自由的日 子没过几天，惨痛马上就随之而来。2000年互联网泡沫破灭，大批公司倒闭，许多人头天还上着班，第二天桌上就放一信封，里头结了一月工资，被裁员了。我 们自然也失业了，那时候刚18岁，以为这行业就跨了，特别迷茫。那种感受特别深刻，以至于我到现在也忘不了，经常告诫初入职场的同事，要随时做好明天公司 就倒闭的准备。

回想那段经历，唯一比较幸运的是比较早接触了技术。当时没有人指点，就是纯野路子，各种瞎学。创业这事黄了之后，我也没去上大学，继续捣鼓编程。我算是被互联网伤过的人，后面找工作一直很纠结，直到2003年底进了新浪，才算是真正成为了正规军。

技术人攻略：如果不考虑现实因素，你想做什么事？你的兴趣爱好是什么?

无论是否考虑现实，我都会做和技术相关的事。中学时期学了6年日语，深受日本文化影响。日本人强调工匠精神，我很欣赏把一个东西从无到有，

并且很精细地做出来那种过程，这可能是我喜欢做技术的原因。但严格说来，我不是死抠技术的人，因为技术始终要服务于产品。

我是野路子出身，属于非典型程序员，喜欢折腾各种各样的新鲜事物。在豆瓣的时候，就喜欢揽事儿，公司自由度也高，往往白手起家倒腾出一个部门，例如CODE、商务开发、东西、Anti-Spam，包括豆瓣读书最早的Kindle版本，都是从无到有做出来的。

除了工作，业余时间我还掺和了很多外界的闲事，搞出了“吃喝玩乐在北京”豆瓣小组，参与国内最早的比特币项目，帮搞艺术的朋友做3D。我身上流着创业的血液，根本闲不住，凡事只要新鲜、好玩，我就愿意做。

我的兴趣爱好很广泛，喜欢踢球、写代码、参加各种线下活动。我非常喜欢生人社交，可能莫名在哪儿看见一个帖子组织聚会，就会去看看。我很愿意认识完全陌生的人，了解完全没有关注过的领域。因为这个原因，交了很多跨界的朋友。我始终觉得，人需要对外界保持一定的新鲜感和敏感度，广泛接触的新事物反刍回来能帮你做好现有的事。

技术人攻略：你兴趣爱好这样广泛，怎么平衡深度和广度的问题?

我一直觉得人的发展是广广深深的过程，广度打不开，深度也会受限制。因为可能挖着挖着就遇到一个特别硬的石头，这时候需要往旁边走，把眼界拉开，才知道应该如何去深。深度这件事在我看来，不是你把MySQL的源码研究得特别透，就叫有深度。我所理解的深度是对整个行业和生活的看法，先要广泛地接触这个世界，然后再讨论更深层的东西。

我们的时代发展得太快，有可能你今天研究的所谓深度，就是明天要淘汰的东西。还是举MySQL的例子，很多公司确实缺资深DBA，把MySQL 研究得特别深当然很好，但你能研究到多深首先是个问题。其次，这个时代光会MySQL也不够用了，还需要了解分布式计算和云计算，停留在原先那个深度上显 然是不够的。如果你早知道Hadoop是下一个时代的数据计算框架，就应该把精力花在新时代的事物上。当然想把Hadoop研究得深，也离不开广度，例如 你需要知道如何把一个算法切成所谓的MapReduce计算方式。反正在我看来，不能一味研究深度，必须得广广深深地往前走。

我自己一直不太敢谈深度，因为始终觉得人外有人。我会的这点东西很皮毛，所谓深度也就是做了多年技术，对于如何写出好的代码，和如何让一

堆工程师一块儿写出好的代码这件事，有浅浅的研究，就算是现在的小优势吧。

原文链接：http://blog.segmentfault.com/devlevelup/1190000000625497